AFIT/GCE/ENG/91D-01

IMPLEMENTATION OF AN OBJECT-ORIENTED FLIGHT SIMULATOR D.C. ELECTRICAL SYSTEM ON A HYPERCUBE ARCHITECTURE

THESIS

Guy R. Booth
Captain, USAF

AFIT/GCE/ENG/91D-01

AFIT/GCE/ENG/91D-01

IMPLEMENTATION OF AN OBJECT-ORIENTED FLIGHT SIMULATOR D.C. ELECTRICAL SYSTEM ON A HYPERCUBE ARCHITECTURE

THESIS

Presented to the Faculty of the School of Engineering

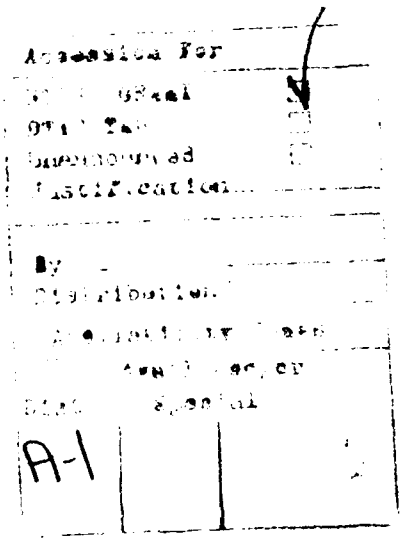
of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Engineering)



Guy R. Booth, B.S.

Captain, USAF

December, 1991

Approved for public release; distribution unlimited

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE
IMPLEMENTATION OF AN OBJECT-ORIENTED FLIGHT SIMULATOR D.C. ELECTRICAL SYSTEM ON A HYPERCUBE ARCHITECTURE

5. FUNDING NUMBERS

6. AUTHOR(S)
Guy R. Booth, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENS/91D-01

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Capt Stuart Bishop
AL/HRAD
Williams AFB, AZ 85240-6457

10. SPONSORING MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Software Engineering Institute developed an Object-Oriented Paradigm for Flight Simulators based on the concept of mapping the behavior of physical objects from an aircraft into an object-oriented software architecture. This mapping is a "semi-formal" method that maps objects to a hierarchy that has three logical layers: objects, systems, and executives. The paradigm was developed with the idea of implementing the derived simulation design on a parallel or distributed computer architecture, but no explicit design features are provided for implementing the design on a parallel computer.

This research addresses the issue of determining what extensions (if any) are required to implement a parallel version of the D.C. Electrical System Simulation (DESS) that the SEI developed as an example on using their paradigm. The parallel DESS design is implemented and tested using Ada on an Intel iPSC/2 Hypercube. An analysis of the performance of the simulation is presented, and some conclusions are made about implementing a parallel design based on the SEI Object-Oriented Paradigm for Flight Simulators.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parallel Processing, Object-Oriented, Flight Simulation, Parallel Computers	147
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Acknowledgments

One of the enjoyable parts of writing a thesis is getting to thank everyone that helped you get the task done, and there are several people to whom I owe a hearty "thank-you."

First, I owe a special thanks to my AFIT instructors. When I first got to AFIT I had no idea what an abstract data type was, much less what an object-oriented design was, and I was at best an average "hacker" when it came to writing code (in FORTRAN, BASIC, or sometimes C). My idea of software engineering was flow charts and no goto statements. Through the efforts of all my AFIT instructors I learned what an abstract data type is, what an object-oriented design is and how to apply "software engineering" principles to the design of both of them.

I owe a great deal of thanks to my thesis committee members. Dr Thomas Hartrum, Dr Gary Lamont and Maj William Hobart. All contributed significantly to this thesis effort and my knowledge about the topic of parallel computing. Maj Hobart's course on Parallel Computer Architectures and Dr Lamont's course on Parallel Algorithms were excellent foundations from which to build my thesis work, and Dr Hartrum's assistance helped to keep my thesis work focused on the research issues. MAJ Eric Christenson provided valuable insights into object-oriented simulation and using Ada, and Rick Norris more than once got me out of some tough spots with problems getting things working right on the hypercube.

I owe a sincere thank-you to the other GCE-91D, GCS-91D and GE-91D students who did research work using the AFIT Hypercube. Their willingness to share "the cube" when I needed to run my simulations is appreciated. You where all true gentlemen and scholars (except for the time one of you had all eight nodes of the cube all weekend!).

Lastly I would like to thank my wife, Jennifer, for her wonderful support during this past eighteen months. I could not have made it through the AFIT program without her encouragement, love and prayers. She is always my biggest encourager, and I thank my Lord for such a wonderful partner for life.

Guy R. Booth

Table of Contents

List of Figures

List of Tables

Abstract

The Software Engineering Institute developed an Object-Oriented Paradigm for Flight Simulators based on the concept of mapping the behavior of physical objects from an aircraft into an object-oriented software architecture. This mapping is a "semi-formal" method that maps objects to a hierarchy that has three logical layers: objects, systems, and executives. The paradigm was developed with the idea of implementing the derived simulation design on a parallel or distributed computer architecture, but no explicit design features are provided for implementing the design on a parallel computer.

This research addresses the issue of determining what extensions (if any) are required to implement a parallel version of the D.C. Electrical System Simulation (DESS) that the SEI developed as an example on using their paradigm. The parallel DESS design is implemented and tested using Ada on an Intel iPSC/2 Hypercube. An analysis of the performance of the simulation is presented, and some conclusions are made about implementing a parallel design based on the SEI Object-Oriented Paradigm for Flight Simulators.

IMPLEMENTATION OF AN OBJECT-ORIENTED FLIGHT SIMULATOR D.C. ELECTRICAL SYSTEM ON A HYPERCUBE ARCHITECTURE

I. Introduction

1.1 Background

The U.S. Air Force uses flight simulators extensively to train flight crews. The systems currently being used to implement these simulators are typically large distributed or uniprocessor systems. These simulations are designed using performance data from an aircraft manufacturer, and are modeled using functional decomposition. Each software component of the simulation models a function of the aircraft such as calculating yaw, pitch and roll moments for the aircraft, or airspeed and Mach number.

Flight simulations for crew training are real-time systems running at update rates of 15–60 Hertz. Fifteen to sixty times per second the state of the simulated aircraft must be calculated, and all systems updated. Since the simulation software is large and complex[1], the software must be run on fast super-mini or larger computer systems to get the required 15–60 Hz real-time performance.[2] For some simulations, the software is divided into large-grain tasks that can run on two or three separate processors running in parallel. This technique is used to increase performance to meet real-time requirements.

Concurrency between flight simulators and the actual aircraft being simulated can be difficult to maintain because changes in aircraft systems, such as the addition of new equipment, are not easily added into a functionally-designed simulator. Changes to one functional module in

[1]The size and complexity are due to the dynamic system models used for aircraft and the large number of state variables maintained in the simulation.

[2]These update rates are based on those seen in systems such as the F-16, F-15 and A-10 simulations used for training systems research at AL/HRA, Williams AFB, Arizona.

the simulation typically require other unexpected changes in other functional modules because of the coupling of functional components in the simulation, and changes may be required in several separate packages to incorporate the new system.

Object-oriented design (OOD) and programming (OOP) can aid in solving many concurrency and implementation problems of flight simulators because object-oriented designs isolate changes to a single object or group of objects. Functional designs can isolate changes to a single functional module if the module is loosely coupled with other modules, but object-oriented design and implementation appear to be better suited to building loosely coupled systems in which data and the methods that operate on that data are grouped together as an object. Since object-oriented design and implementation employ information hiding and encapsulation, changes to a single object do not usually require changes to other components throughout the system [8, 7]. Modifications are limited to specific objects, and few, if any, modifications are required in other objects within the system.

OOD provides the capability to deal with very complex software systems in a consistent manner that models real objects more directly than functional, or data-oriented designs can [8, 7]. This is another reason that OOD is being used in developing simulations [24, 22].

The Software Engineering Institute, a federally funded software consortium at Carnegie Mellon University, has developed a paradigm to map "real" objects from an aircraft into OOD and OOP objects for an object-oriented flight simulation [30, 29]. The paradigm is well defined and its authors have used the paradigm to implement a uniprocessor simulation of a direct-current electrical system for a flight simulator [29]. The SEI OOD Paradigm is the result of work done in the Ada Simulator Validation Program (ASVP), a research and development effort by two aerospace contractors to redesign and implement subsets of two existing flight simulators in Ada. The SEI OOD Paradigm and the software architecture developed by the SEI have received acceptance and are being used by contractors involved in full-scale simulation developments [30].

"One of the design goals of the paradigm was to facilitate spreading the work load over multiple processors" [30:45]. Some ideas for doing this are presented in the paradigm report; however, the developers have not implemented or tested any of the presented ideas [30:45]. The paradigm does not present any heuristics or methods for dealing with the parallel programming problems of non-determinism, load-balancing, and communications overhead. These are important factors that must be considered in mapping any software design to a parallel computer architecture [10, 19, 1]

The simulation objects developed using the SEI OOD Paradigm display the same concurrency seen in the real aircraft, i.e., the engine components run in parallel with the electrical system components, which run in parallel with the avionics components, etc.. Also, components (objects) within a single system appear to have some parallelism. The parallelism within a single system is a smaller grain parallelism than the system level parallelism, but may afford an additional increase in performance of a parallel implementation of the simulation.

If a method can be derived for implementing the SEI OOD Paradigm on a parallel architecture, then the task of implementing any new SEI OOD designs on parallel architectures can be simplified, and development time and cost for U.S. Air Force flight crew simulators can be reduced.

1.2 Problem Definition

Can the SEI OOD Paradigm be extended to provide a method of implementing a flight simulator on a distributed memory parallel computer architecture?

1.3 Assumptions

1. The SEI OOD Direct-Current Electrical System Simulation is a "correct" implementation (i.e., it accurately simulates the model on which the electrical system design is based), and it is a good example of a typical design derived using the SEI OOD Paradigm. This assumption

is based on the SEI's choice to use this simulation to demonstrate the application of their design paradigm. Thus, it is a good baseline to use for research on developing a parallel implementation of a flight simulator that is based on the SEI OOD Paradigm.

2. The SEI simulation is deterministic; given the same inputs, it always generates the same results. If the sequential simulation is deterministic, then the output of the parallel implementation can be compared against the sequential implementation to validate the parallel simulation. If the sequential simulation is not deterministic, then validating the correct implementation of the parallel version of the simulation may not be possible.

3. The SEI simulation has objects that exhibit large or medium-grain concurrent behavior.[3] If the objects in the simulation do not exhibit large-grain or medium-grain concurrent behavior, then fine-grain parallelism will be the only type of parallelism possible. However, it has been shown that the iPSC/2 Hypercube architecture is not well-suited for implementing fine-grain parallelism due to the high ratio of communication time to computational time [28:63]. Thus, attempting to implement a fine-grain parallel design of the SEI simulation may not result in any speedup of the simulation; it could run much slower than the sequential version due to communications overhead.

4. The SEI OOD simulation can be implemented and run in the memory space available on the iPSC/2 Hypercube. Currently, each node of AFIT's Intel iPSC/2 Hypercube has twelve megabytes of memory. If the simulation requires a larger system than the AFIT iPSC/2 Hypercube, then another hypercube system will need to be found that has more memory, or the simulation implementation will need to be optimized for minimal memory usage.

5. All the simulation objects are persistent [8], i.e. all objects are created during the initialization of the simulator, and no objects are deleted or removed during the simulation. This is

[3] Large-grain concurrency is parallelism at the outer level of program control, small-grain concurrency is parallelism at the instruction or logical operation level, and medium-grain parallelism is between these extremes [14:8].

consistent with the objects being simulated; the electrical components in the real aircraft are not deleted or destroyed. They are persistent objects.

6. All object communication paths are known during the initialization phase of the simulation, and communication connections between objects are static. This is a result of the persistence of the objects noted above, and the physical connections of the components in an aircraft.

7. All objects in the simulation have the same basic behavior: they receive input messages from other objects, they calculate a new state based on these inputs, and their new state information is sent as an output to other objects. This is the basic object behavior that is used to build a model of the parallel objects of the simulation.

8. Dynamic task scheduling for objects is not required. All objects are allocated to processors at initialization time, and the objects are not moved between processors, but messages are. The issues of *optimal* task scheduling and dynamic load balancing are not a major point of investigation in this research. This should not be a limiting factor on the research value of this thesis because little work has been done on object-oriented, parallel, time-driven simulations (see Chapter II); and the topics of dynamic scheduling of objects and tasks is an area of research being investigated by other researchers.[4]

1.4 Scope

This research effort addresses how to implement a simulator designed using the SEI OOD Paradigm on a distributed, parallel, multiple instruction, multiple data (MIMD) computer.

The SEI OOD Direct-Current Electrical System Simulation is used as the basis to derive a parallel design that is implemented in Ada on the AFIT Intel iPSC/2 Hypercube parallel computer system.[5]

[4] Other AFIT students addressing this issue: JoAnn Sartor [40] and Andrew McNear [33].

[5] Ada is used as the implementation language since the SEI DC Electrical System source code is in Ada and an object-based language such as Ada will aid in implementing a parallel version of the SEI's simulation.

1.5 Standards

Speedup of a parallel program is typically measured by comparing the performance of the best sequential algorithm against the performance of the parallel algorithm [19, 1]. To measure the speedup of the parallel OOD simulation, a comparison of the execution time of the parallel, object-oriented simulation versus the sequential SEI simulation execution time is done.

The simulation executes a single iteration by sequencing through a series of operations that pass state information between the objects in the simulation. The number of simulation iterations is varied and the execution time of both simulators measured. The results are presented graphically and tabularly to show the performance of the parallel simulation.

1.6 Approach/Methodology

The parallel simulation is developed by first analyzing the SEI OOD simulation software, and determining the concurrency in the design. Since a key benefit of the SEI OOD Paradigm is the well-defined software architecture, the overall software architecture of the original simulation is maintained in the parallel simulation with extensions added to provide the necessary mechanisms for parallel execution. A large part of the original SEI simulation software is reused in the parallel version.

The parallel extensions are designed after the concurrency analysis is completed. The new design is implemented on the iPSC/2 using Verdix Ada. Changes are made to the various packages of the simulation, and each change tested prior to making the next change. This is done to control the number of errors generated due to new code, and to make sure the changes implement the desired behavior.

Once all the modifications to the simulation are complete, the simulation is tested to validate that it has a deterministic performance, and the results of this testing are compared with the sequential simulation running on the iPSC/2 host processor. This provides some level of confidence

that the simulations behave the same for the same inputs. No attempt is made to do a formal logical proof (verification) that the simulations are identical; that is beyond the scope of this thesis effort.

Next, a series of tests is run to compare the execution times of the parallel simulation against those of the sequential simulation to measure the speedup (or slowdown) of the parallel simulation. Various configurations of object-to-processor mappings are tested to determine the effect on speedup for various configurations. The execution times were measured on one, two, four and eight nodes since a hypercube architecture requires that processors be allocated in powers of two, i.e., the number of processors is 2^d where d is the dimension of the hypercube. The AFIT iPSC/2 has a maximum dimension of 3 as it is currently configured.

The original SEI simulation is modified to run on a single node of the iPSC/2 Hypercube, and then the performance of the sequential version is measured. The sequential version is tested on a single node, instead of the host, since the host processor is a multi-user system with a large UNIX-based operating system. Each node is a dedicated, single-user processor that does not have the high operating system overhead seen on the host node.

Last, an analysis of the results of the timing measurements is presented, a performance estimation model is described, and some conclusions based on the performance estimation model are presented.

1.7 Sequence of Presentation

Chapter II of this thesis is a literature review of object-oriented design and programming, object-oriented simulation (OOS), and parallel OOS. Chapter III is a description of the SEI OOD Paradigm and the DC Electrical System Simulation. Chapter IV is an analysis of the SEI OOD Paradigm and a description of what extensions were added to the SEI Paradigm to implement a parallel simulation on a distributed computer architecture. Chapter V presents the implementation

of the parallel simulation. Chapter VI documents the testing results, and Chapter VII is an analysis of the testing results. Chapter VIII contains conclusions about this research effort and makes some recommendations for future research.

II. Literature Review

2.1 Introduction

This chapter describes the current state-of-the-art in object-oriented simulation (OOS) and how OOS is being implemented on parallel computer systems.

In 1989, the Department of Defense identified modeling and simulation as one of the twenty-two critical technologies in the United States[16:A–55]. Antonio Guasch, 1990 editor of *Proceedings of the Society for Computer Simulation Multiconference on Object Oriented Simulation*, states that

> Object-oriented methodologies and programming languages are becoming increasingly important in software engineering in general, and in simulation in particular. Object-oriented design has assumed in the last few years an important role in such simulation areas as discrete-event simulation, distributed simulation, graphical interfaces for simulation and knowledge simulation. Object-oriented design is a natural approach to modeling and studying a world comprised of objects. It is our belief that the ideas behind this methodology will ultimately spread into most of the simulation community. Practical aspects have limited the widespread use of object-oriented simulation-based systems. However, new object-oriented programming implementations that reduce the amount of computer resources needed to support them will help to overcome some of the present limitations [22:1].

Parallel computer architectures (hardware and software) were also identified by DoD as a critical technology [16:A–33]. Simulations are computationally intensive, and recent development of cost effective parallel computers provides the opportunity to broaden dramatically the range of problems that can be simulated [38]. Parallel object-oriented simulations[1] provide a means to implement complex simulations that can be executed in an acceptable amount of time [2, 6, 27].

This review is a summary of the professional literature of published sources from 1987 to date on the subject of parallel object-oriented simulation. The information presented was gathered from a search of the Defense Technical Information Center, Compendex Plus, and Dialog Aerospace databases at the Air Force Institute of Technology Academic Library. The scope of the review

[1] Also referred to as distributed OOS in the literature.

is limited to parallel object-oriented simulations that are not knowledge based, and it does not address simulation using artificial intelligence (AI) languages such as object-oriented Common Lisp or DEVS-Scheme.

The review starts with a short background of object-oriented design (OOD) and object-oriented programming (OOP) and introduces the key terms and concepts used in the rest of the review. Following this review of OOD and OOP, object-oriented simulation, and then parallel object-oriented simulation are reviewed. Lastly, some conclusions are presented based on the current development of object-oriented simulation and parallel simulation.

2.2 Object-Oriented Design and Programming

Object-oriented design is a methodology that has been popular in the computer simulation and artificial intelligence communities for some time. Since more efficient object-oriented programming and support environments are becoming available, OOD and OOP are gaining wide acceptance in the general software community because they aid in developing modular and reusable software [6:1–2].

Object-oriented design is the method that leads to an object-oriented decomposition and provides the capability to deal with very complex software systems [8:1–23]. In this method of design, data and its associated functions are encapsulated in self-contained units called *objects*. Each object has ***attributes*** that describe its *state*, and the object provides *services* or *methods* that allow its state to be determined or changed. Each object is part of a *class* of objects that have similar attributes and services. These objects communicate with each other by passing *messages*, and each object exhibits behavior in response to the messages it receives. [6, 7, 32].

Object-oriented programming is used to implement an object-oriented design. Using an object-based or object-oriented language such as C++, Smalltalk, Object Pascal or Ada, a programmer can implement an object-oriented design in a straight-forward manner [8, 32]. An object-

oriented programming language is not required to implement an object-oriented design, but the implementation is much easier for the programmer when an OOP language is used [13].

Charles Herring from the U.S. Army Construction Engineering Research Laboratory states that, "Object-oriented programming is replacing structured programming as the dominant software paradigm. This movement will have a great impact on the design and production of simulations" [24:59].

2.3 Object-Oriented Simulation

Lomow and Baezner [32] describe how object-oriented simulation builds on the principle that a system design is based on objects. The objects are representations of the elements that compose the system. Some objects in an OOS execute independently and concurrently with other objects. These are called *entities*, and they model the physical processes of the simulated system. *Events* cause a change in the state of an object in the simulated system. Entities schedule events for each other and either synchronize the actions of two or more entities or pass information between entities. All the actions of entities and the scheduling of all events are tied to a logical clock called the *simulation clock*. Each event is tied to the logical simulation clock by means of a scheduled event time, and the event time corresponds to the actual time in the physical system when the corresponding physical event occurs [32].

Languages such as ModSim [24], Sim++ [2] and Ada [29] are being used to implement object-oriented simulations. ModSim and Sim++ are supersets of the Modula-2 and C++ languages, respectively, and are designed specifically to support object-oriented simulation [24, 2]. Ada is a general purpose object-based programming language [13, 8, 7] developed initially to support complex embedded systems programming [12]. Since Ada is the approved High Order Programming Language for DoD, it is being used to implement various simulation systems. The Ada language contains some support for object-oriented programming but has some notable deficiencies [13]. Ada

does not incorporate the object-oriented design concepts of inheritance and dynamic binding. To support object-oriented simulation using Ada, Corbin and Butler [13] have developed a toolkit to provide the facilities to create and manipulate objects and to provide support for other useful features using Ada in simulation. Using their toolkit, no extensions to Ada are necessary to implement an OOS, and all programming is done using the standard Ada language.

Members of the technical staff at the Software Engineering Institute, a federally funded software consortium, have developed an object-oriented paradigm for building flight simulators using Ada [30], and they have developed an electrical system simulation for a flight simulator using the paradigm [29]. The paradigm provides a structured method to map "real" physical objects from an aircraft to software objects in a flight simulator, and their paradigm does not require any extensions to Ada to implement the simulation on a uniprocessor system. A full description of the SEI OOD Paradigm and the DC Electrical System Simulation is presented in Chapter III.

Robert Doyle at the Rockwell International Science Center has implemented a communications simulation using several OOP languages (Smalltalk, Objective-C, C++, and Turbo Pascal with Objects) and non-OOP languages (C, Common Lisp, Simscript II.5), and he has measured the simulation performance for each language. [17]. He found that the non-OOP simulation using C was the fastest simulation (60 CPU seconds for a simulated time of 10 units). However, he found that the object-oriented C++ implementation ran in nearly the same time (62 CPU seconds for 10 units). He also found that all the languages tested yielded acceptable performance. His conclusion was, "Object-oriented techniques may be successfully applied to simulation with little if any performance penalty. However, some care must be taken in selecting the implementation vehicle (or programming language)" [17].

2.4 *Parallel Object-Oriented Simulation*

Parallel computer architectures can be classified as either loosely coupled or tightly coupled [14]. Loosely coupled systems have their memory distributed among each processor as local memory. If all processors share a common global memory, then the system is a tightly coupled system. Currently shared memory, tightly coupled architectures are limited to systems of only a few hundred processors, but distributed memory, loosely coupled systems are available with thousands of processors [14, 16].

Complex object-oriented simulations require several objects, and large numbers of objects require a large amount of processing and memory resources; parallel computer architectures can provide the resources needed to support complex object-oriented simulations [3]. William L. Bain and Shala Arshi have shown that a complex, object-oriented simulation model can be implemented to run efficiently on an iPSC/2 Hypercube [4]. Bain and Arshi used the *Interwork II* concurrent programming toolkit to implement an object-oriented, discrete-event simulation of a parallel computer system on the iPSC/2 Hypercube, a distributed memory, multiple instruction, multiple data (MIMD) computer system.

Using a distributed parallel architecture to implement a simulation will usually result in a speedup when compared to the time that the same simulation requires on a uniprocessor system [20, 23, 35]. Hartrum and Donlan have shown a speedup by using distributed simulation for a battle-management simulation of a ballistic missile defense system [23], and Nicol was able to achieve a speedup for another battle simulation [35]. Fujimoto conducted a series of tests on the speedup achieved by several different distributed discrete-event simulations [20]. In comparisons with uniprocessor-based event simulations, he showed that significant speedups can be achieved for *most* simulation workloads. However, he did find that some parallel simulations ran *slower* than the equivalent uniprocessor simulation.

Research in parallel simulation has primarily focused on discrete-event simulation. The two primary methods of discrete-event simulation discussed in the literature are the Time-Warp optimistic approach [32, 5, 44, 21, 31, 35] and the Chandy-Misra conservative approach [9, 34, 39, 20, 36, 43]. Parallel, time-driven simulation has not received as much attention in the literature [45, 37, 11, 23]. No published sources were found on object-oriented, parallel, time-driven simulation.

2.5 *Review Summary*

Object-Oriented design and programming are becoming more important in the software engineering community because OOD and OOP provide a method to model accurately the world as people see it — a world of objects [22, 24, 8].

The SEI OOD Paradigm was developed to aid in mapping "real" objects into OOD and OOP objects for object-oriented flight simulations [30, 29]. The paradigm is well defined and has been used to model real systems of objects on uniprocessor systems; however, the paradigm has not been shown to work for *parallel* object-oriented designs.

It has been shown that distributed memory parallel systems can be used to implement complex object-oriented simulations [3, 4]. Usually the simulation will execute faster on the parallel system, but not always [20].

This research effort will investigate whether the SEI OOD Paradigm can be extended to develop a parallel, time-driven, object-oriented simulation that executes in less time than the equivalent uniprocessor simulation. The SEI's DC Electrical System Simulation (DESS) will be used as an example to evaluate what extensions are required to the SEI OOD Paradigm to implement such a simulation. If the paradigm can be extended and implemented effectively, then a structured method of implementing a parallel, time-driven, object-oriented, aircraft simulator can be developed based on the extended SEI OOD Paradigm.

III. Overview of the SEI OOD Paradigm for Flight Simulators

3.1 Introduction

The 1988 SEI report, *An OOD Paradigm for Flight Simulators, 2nd Edition*, describes the design of a reusable architecture for the flight simulator software domain [30]. In the report, a design for an engine system is presented to illustrate application of the design paradigm, and the engine system software is developed to the Ada package specification level.[1] The bodies of the engine system objects are not developed in the report.[2]

The 1989 report, *An Object-Oriented Solution Example: A Flight Simulator Electrical System*, demonstrates the application of the paradigm to derive a design for a DC electrical system of a flight simulator [29]. The Ada package specifications and bodies for the DC system are developed in this report along with a test driver for the simulation. In the following paragraphs an overview of the SEI paradigm is presented to provide a basic familiarity with the key concepts of the SEI Paradigm. The description is extracted from [30] and [29], and the reader should refer to these reports for a detailed description of the paradigm. Spicer's thesis, [41], describes the paradigm from a software reusability viewpoint.

3.2 A Unique OOD Paradigm

The SEI OOD Paradigm was designed with two basic goals: to eliminate nested implementations of objects[3] and to simplify dependencies among objects [30]. The paradigm is a model for implementing systems of objects to meet these goals. In his thesis, *Mapping an Object-Oriented Requirements Analysis to a Design Architecture that Supports Design and Component Reuse*, Spicer

[1] Ada specifications define the object types and procedures that are visible to other packages. Ada package specifications can be compiled but not linked without package bodies.

[2] Package bodies are required to implement executable Ada programs.

[3] Nesting of objects occurs when objects are decomposed into groups of other objects. These objects are decomposed into still other objects, etc..

notes the following characteristics of the SEI OOD software architecture that make it unique when compared to other OOD architectures.

1. The architecture consists of logical layers replacing the usually nested objects found in a composition hierarchy. Though the control flow follows the hierarchy, data generally flows across the hierarchy, that is, data may pass directly between different major software units without going up and down the hierarchy.
2. Templates are used for instantiating the architecture for new applications within the flight-simulator domain.
3. Connectors are used to connect objects. This reduces coupling and renders the objects themselves more reusable since there are no direct dependencies (Ada "with"ing) between objects [41:2–9].

3.3 The Paradigm Components and Layers

The paradigm defines an architecture with three logical layers: the object, the system, and the executive layers [41:3–5]. The following are excerpts from [30] and [29] that describe the logical layers and the components within the layers.

> The fundamental units of the paradigm are *objects* and *connections*. *Objects* map to real–world entities. An *object* is implemented as a math model that maps the environmental effects (inputs) on the object to the object's outputs, given the attributes of the object and its operational state. The implementation isolates individual effects. Also, an object is not aware of its connections to other objects.
>
> A *connection* is the mechanism for transferring state information between objects. Processing a connection involves reading the state of some objects on the connection and broadcasting to others [30:11].
>
> Connections are procedures which read the state of one object and write that state to another object...
>
> Connections are classified as either executive or system level connections. Executive level connections are those between systems ... System level connections are those between objects within a system.
>
> System level connections read state information from an object in the system and write the information to another object in the system. The connections are owned by the system. When the system is called to update itself, all the system level connections are gated (or processed); when complete, the system is considered updated.

> Executive level connections read state information from an object in one system and write the information to an object in another system. ... When the DC Power System is updated, the flight executive connections to the system are gated, then the system is called to update itself [29:8].

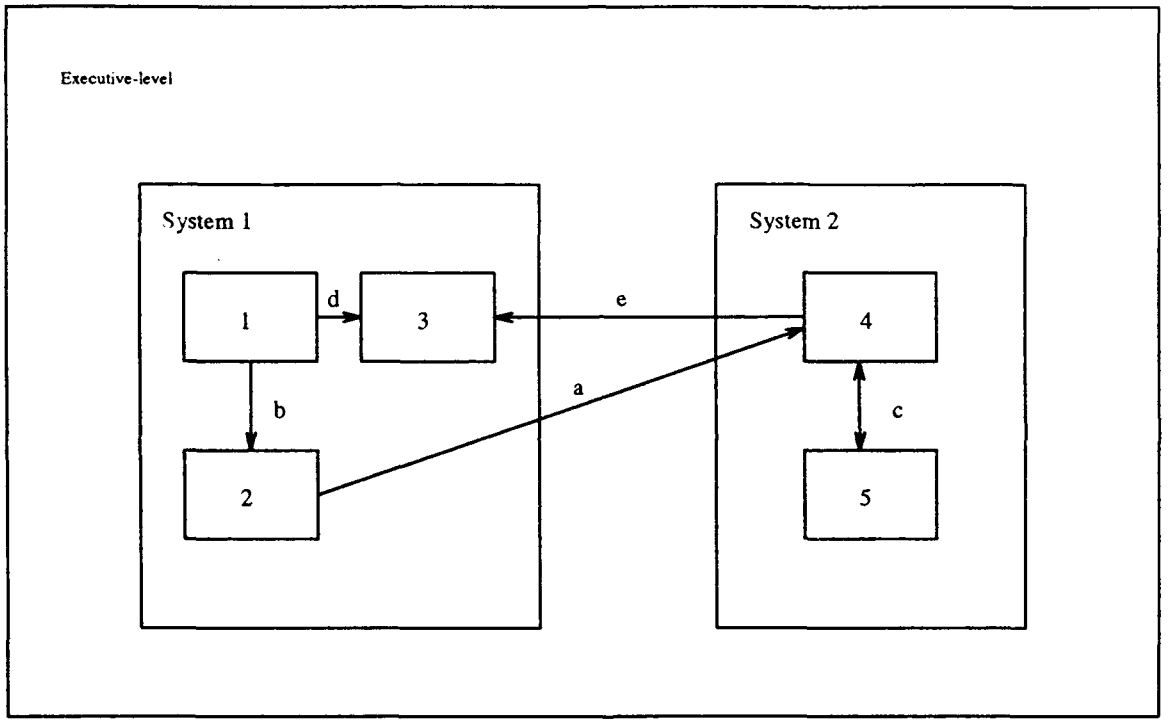
The system level is defined by the objects within a system and the connections between the system objects (intra–system or system-level connections).

Above the layer of objects and systems is the executive layer. "An *executive* controls the update of a set of systems compiled together running on a single processor. The paradigm assumes that there will be more than one set of systems and that multiprocessing will be involved" [30:11]. Thus, the paradigm implies that parallelization is to be implemented at the system level, which is a course-grain level of parallelization. (For this research, medium-grain parallelization within a system is investigated. See Chapter IV for more details on the levels of parallelization.).

> Communication between executives is handled by an abstraction called a *buffer*. A *buffer* is some means of sharing data among separately compiled software. The paradigm makes no assumptions about how the operating system transfers data or how executives on separate processors are invoked.
>
> At all levels, updates are accomplished by gating (or processing) the appropriate connections. The levels discussed in the paradigm are *system* and *executive*. A *system* is an aggregation of objects and the connections among those objects. An *executive* is a set of systems and all connections that cross system boundaries, i.e., connections between objects in different systems. Figure 1 shows views of an executive, two systems, and several objects and connections [30:11,12].

Each object class is represented by a single object manager, and access to each object is handled through the methods defined by each object manager.

> The object manager defines the *operational state* of the object. The operational state refers to those characteristics which may change with time, e.g., the degree of charging or discharging of a battery, the setting of a switch, malfunctions, or aging effects on various components.
>
> The object manager allows the object's *environmental effects* to be placed on the object. The environmental effects are external object states which are required by the



Executive is: System 1, System 2, and connections a and e
System 1 is: Objects 1, 2, and 3, and connections b and d
System 2 is: Objects 4, 5, and connection c

Figure 1. Object Diagram Example [30:12]

object to determine its state.

> The object manager implements the math model for the object. The math model is implementation dependent. The object managers use the math models to map the object's inputs to its outputs. The object manager produces the *outputs* available from the object. The outputs are generated by the math model, using the environmental effects placed on the object and any additional constraints imposed by the attributes and the operational state of the object. The math model may be invoked when the environmental effects are placed on the object or when outputs are read from the object. This is an implementation level decision left to the system designer; it is not defined by the paradigm.
>
> The actual instances of the objects are stored in system aggregates. An aggregate allows named access to the objects in a system; no procedure call is required to retrieve the object. The aggregate is not denoted on design specifications, but is an essential part of the implementation of a system and its objects [29:9].

The SEI paradigm defines abstractions that can be used to implement an object-oriented simulation in a consistent manner. The object managers define classes of objects, and provide the methods that can be applied to the objects of that class. Interaction of objects is accomplished by gating the executive and system-level connections between objects. The paradigm does not stipulate that a connection must have a single source or destination. Also, it does not stipulate that the flow of information through a connection must be only in one direction, i.e., state information may be transferred bi-directionally through a connection.

Objects defined by the SEI paradigm are not *active* objects or agents; objects do not request services directly from other objects. Connections are the *active* agents of the simulation. When a connection is gated (or processed), the connection requests services of the object it reads and the object it writes. Thus, the gating of connections is the active part of the simulation. By gating more than one connection at a time in a system, some degree of parallel execution may be achieved in the simulation. However, dependencies in the processing order of connections may limit the potential parallelism. (Chapter IV describes how this parallelism can be used to implement a scalable parallel simulation using the SEI paradigm.)

Aggregates provide an abstraction for systems in the SEI paradigm. The system aggregates provide a single software unit that contains the instances of the objects in a system. A system

aggregate maintains pointers to several classes of objects that define a system. This simplifies building of systems from objects and provides a single point of access to a system's objects. The concept of the system aggregate providing named access to all the objects within a system is different from the typical method used in object-oriented designs. Typically, the object managers would maintain a list of pointers to the objects of the class defined by the object manager. Then, access to an object would be done through its object manager. In the SEI's paradigm, the object managers are relieved of the task of tracking the object instances. The object managers simply provide the methods for the type of objects they export.

In the following section the SEI DC Electrical System Simulation (DESS) is presented as an example of how the SEI OOD Paradigm can be used to implement a system within a flight simulator. The section is a summary of the information contained in [29]. Please refer to [29] for a full description of the DESS.

3.4 The DC Electrical System Simulation

Figure 2 is a schematic diagram of a DC electrical system that might be seen in a typical aircraft [29:2]. AC power is provided to the transformer rectifier units (TRUs) that convert the AC voltage to a DC voltage. The TRUs are the source of current and voltage for the DC Power System, and they have a load conversion factor (LCF) that represents how effectively the TRUs convert AC to DC power. DC buses carry the voltage and current from the TRUs to the load devices. The buses follow Kirchoff's current and voltage laws in the simulation. Circuit breakers (CBs) connect various components together, and can be switched off (or open-circuit) by using the set_position method provided by the circuit breaker object manager. For the simulation the CBs are considered to have no loss across them.

The DESS objects are of three classes: buses, TRUs and CBs. Each of these classes of objects is implemented with a separate object manager that defines the proper types and methods for these

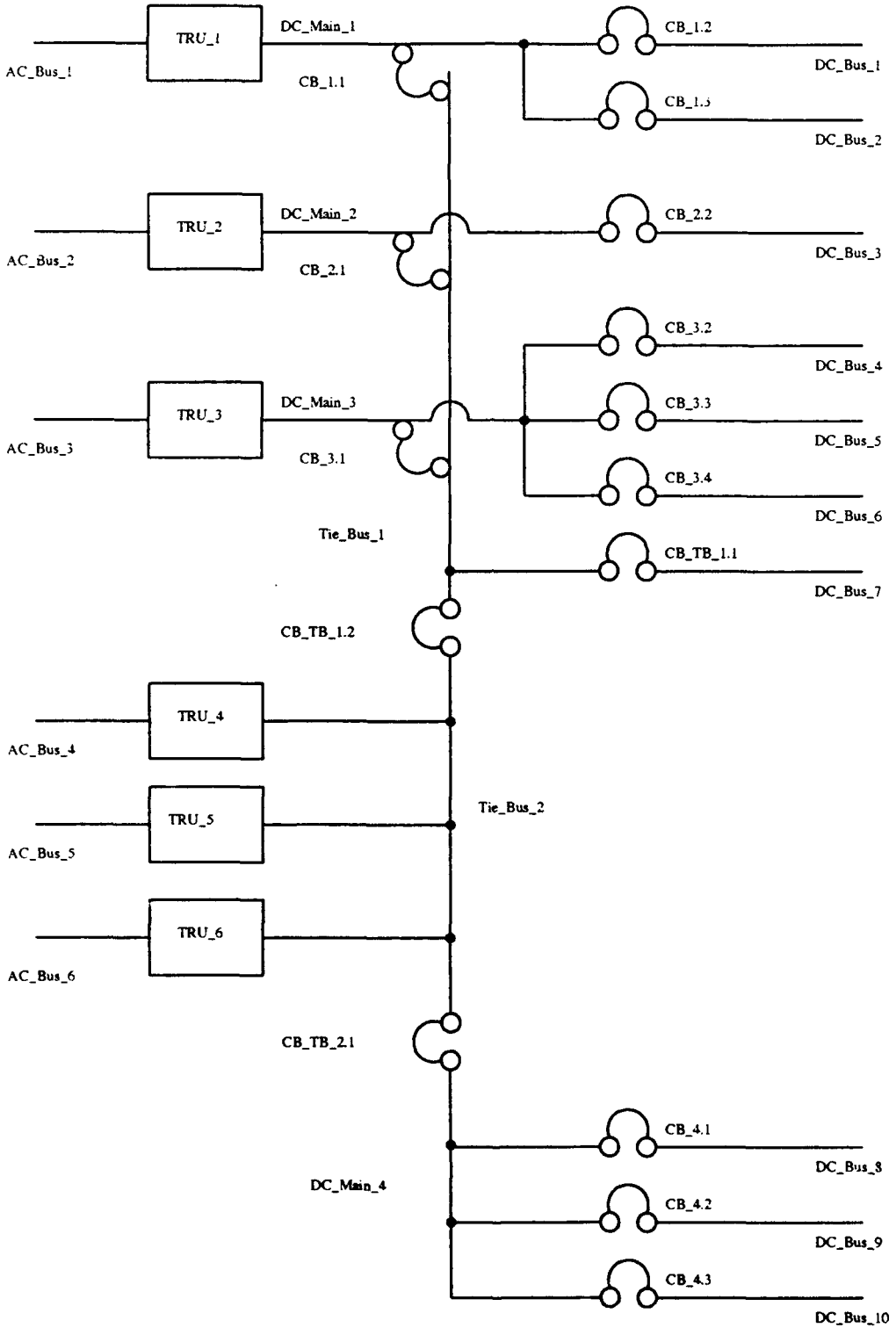


Figure 2. DC Power Circuit Diagram [29:3]

objects. The objects for the design specification are derived by mapping the real-world entities from the circuit diagram shown in Figure 2 to objects as shown in Figure 3.

Figure 3 shows the Dummy System added as an external load to the DC Power System. The Dummy System is composed of CBs instantiated in the Dummy System Aggregate. The AC power is provided by buses in the AC Power System. Six AC buses are connected to the DC Power System's six TRUs, and the six AC buses are instantiated in the AC Power System Aggregate.

The paradigm defines the structural model of the simulation in terms of a graphical design specification such as the one shown in Figure 3 for the DC Power System. The structural model consists of symbols and rules where each symbol has a software template associated with it [29:11]. Thus, the paradigm provides a structured method of mapping real-world entities into objects created using software templates, and the diagram shown in Figure 3 represents a specification of the simulator's DC Power System design. The design specification diagram shows three systems that define the operational environment of the DC Power System. The AC Power System provides voltage and current to the DC Power System, and the Dummy System represents the load on the DC Power System [29:6].

The DC Power System design specification follows the convention that:

- A bus object is between every other pair of connected objects.
- Only two connection points are on each connection; one on a bus object and the other on another object [29:6].

> The solution (or simulation design) models the flow of information directionally. Voltage and LCF flow from voltage sources, such as the generators and batteries, through passive objects (such as circuit-breakers) to voltage sinks, such as motors and lights. Load flows from the voltage sinks back to the voltage sources. Because of this directionality, objects, such as circuit breakers and TRUs, are defined with two sides. Each side contains the information flowing through the circuit to that point. The transfer of information through an object means: obtain the stored information from the opposite side of the object. This convention maintains the proper flow through the

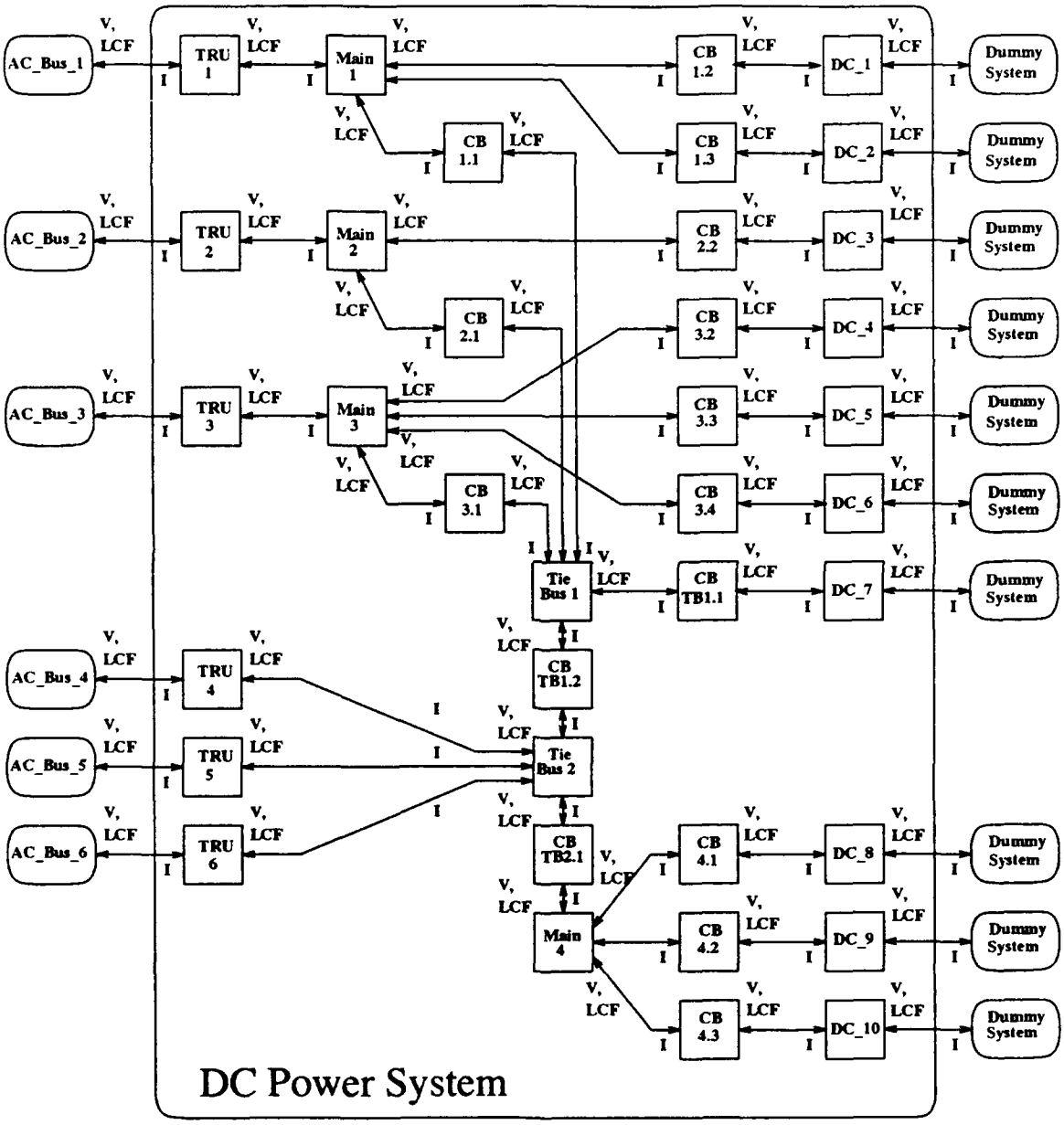


Figure 3. DC Power Design Specification

system. Bus objects, which may be connected to any number of other objects, have as many sides as they do connection points [29:6].

In Figure 3 connections are represented by double-headed arrows. The typical convention for design specification diagrams as shown in Figure 3 is that information is read from the object at the tail of the arrow and written to the object at the head of the arrow [29:8]. However, Figure 3 uses double-headed arrows for notational simplicity. Voltage, V, and LCF data flow from left to right in the figure, and current, I, flows from right to left in the figure. For example, voltage, V, and LCF are read from the AC Power System bus objects and written to the TRUs in the DC Power System. Current, I, is read from each TRU and written to the AC Power System buses.

The connections such as those from AC_Bus_1 to TRU_1 in Figure 3 and from DC_1 to the Dummy System are all executive-level connections. These connections are processed by the executive prior to processing the system connections within the DC Power System. The connections enclosed in the DC Power System roundangle[4] are all system-level connections for the DC Power System.

3.5 Paradigm Software Architecture

The paradigm uses a unique software architecture to implement a simulation design specification constructed from the abstractions of objects, connections, systems and executives. In the following paragraphs, we will use the DESS software from [29] to describe the software architecture derived from the design specification shown in Figure 3. Refer to [29] for a listing of the SEI source code for the DC Electrical System Simulation as implemented from the design specification using the paradigm's software architecture.

A goal of the SEI paradigm is to simplify dependencies between objects [30:8]. The objects created using the paradigm are implemented so they are only dependent on global types [29:12].

[4] A rectangle with rounded corners

```
package Electrical_Units is

     type Voltage is (Floating_Voltage, Zero_Voltage, Available_Voltage);

     No_Voltage : constant Voltage := Zero_Voltage;

     -- Devices like relays need to know if
     -- voltage is available without concern for the level.
     --
     Energizing_Voltage : constant Voltage := Available_Voltage;

     type Current is new Float;
     No_Current : constant Current := 0.0;

     type Load_Conversion_Factor is new Float;
     No_Load_Conversion : constant Load_Conversion_Factor := 0.0;

     --  Needed when device shorts out when current passes the wrong way.
     --
     Max_Load_Conversion : constant Load_Conversion_Factor := 10_000.0;

     --  Permits a function to return all three values.
     --
     type Power_Info is
               record
                       V : Voltage := Floating_Voltage;
                       I : Current := No_Current;
                       Lcf : Load_Conversion_Factor := No_Load_Conversion;
               end record;

end  Electrical_Units;
```

Figure 4. Electrical_Units Package [29:13]

The global types package for the DC Power System is called `Electrical_Units`. The package provides the definition for voltage, current, and LCF types. Voltage is an enumerated type; but current and LCF are floating point values. The Power_Info record contains components for voltage, LCF and current. Figure 4 is a copy of the `Electrical_Units` package.

Each object type has an object manager associated with it. Each object manager provides the methods for manipulating its object type, and each object manager has a similar structure. Each object manager for the DESS provides the following methods:

- `New_{object}`. Returns a new instance of type object.
- `Give_Voltage_Lcf_To`. Gives voltage and LCF to one side of an object.
- `Give_Current_To`. Gives current to one side of an object.
- `Give_Power_Info_To`. Gives power info to one side of an object.
- `Get_Power_Info_From`. Gets power info from one side of an object.

The circuit breaker object manager also provides a method to get the position of a circuit breaker object (open, or closed), and a method to change the position. Figure 5 shows the Ada specifications of the circuit breaker object manager.

Every electrical system object has the attribute `Side_Names`. "In order to simulate energy flow in a system, the sides of an object hold the flow through the system to that point" [29:69], i.e., each side of an object holds state information that represents the energy flow to that point. Energy flow in the DESS is simulated by transferring state information from the side of one object to the side of another object. By gating a connection between two objects, the state information from the side of one object is transferred to the side of another object.

A typical object from the DESS is shown in Figure 6. "The voltage flow through the system to the object is stored at side X. The load flow through the system to the object is stored at side Y" [29:69]. When gating a voltage connection that has side Y as the source of the connection, the object operation `Get_Power_Info_From (a_side => Y)` is applied to the Y side of the object shown in Figure 6. Applying this operation to the Y side of the object causes the voltage information on side X to "flow" through the object. When the voltage flows through the object, the voltage state data is transformed by operations within the object that model the behavior of the type of object being simulated, e.g., a TRU object will convert the voltage on side X from an AC voltage to a DC voltage.

```
with Electrical_Units;

package Cb_Object_Manager is

     package EU renames Electrical_Units;

     type Cb is private;
     type Cb_Position is (Open, Closed);
     type Cb_Rating is new Float;

     type Cb_Side_Names is (Side_1, Side_2);

     function New_Cb (Position : in Cb_Position;
                                 Rating : in Cb_Rating)    return Cb;

     procedure Give_Voltage_Lcf_To (
                               A_Cb : in Cb;
                               A_Cb_Side : in Cb_Side_Names;
                                Volts : in EU.Voltage;
                               Load_Conversion : in EU.Load_Conversion_Factor);

     procedure Give_Current_To (
                              A_Cb : in Cb;
                              A_Cb_Side : in Cb_Side_Names;
                              Load : in EU.Current);

     procedure Give_Power_Info_To (
                              A_Cb : in Cb;
                              A_Cb_Side : in Cb_Side_Names;
                              External_Power_Info : in EU.Power_Info);

     function Get_Power_Info_From (
                              A_Cb : in Cb;
                              A_Cb_Side : in Cb_Side_Names)

                              return EU.Power_Info;

     procedure Give_Position_To (
                             A_Cb : in Cb;
                             Position : in Cb_Position);

     function Get_Position_From (A_Cb : in Cb) return Cb_Position;

private
        type Cb_Representation;  -- incomplete type, defined in package body
        type Cb is access Cb_Representation;-- pointer to a Cb representation

end Cb_Object_Manager;
```

Figure 5. Cb_Object_Manager Package [29:15]

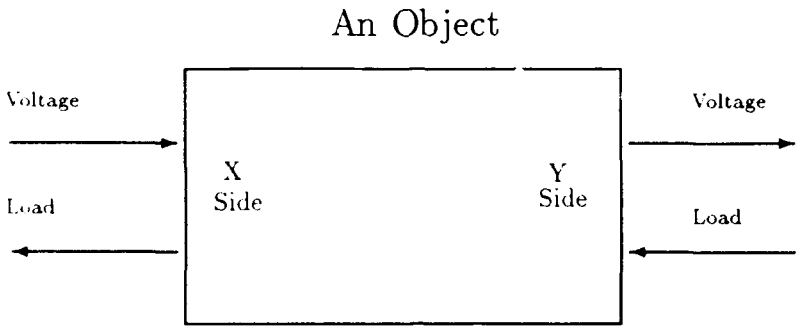


Figure 6. Energy Flow from Object Sides [29:69]

The calculations required to implement the flow of energy through an object can be done when state information is applied to the side of an object or when the state information is read from the side of an object using a method such as the `Get_Power_Info_From` method described above. The paradigm does not specify when these calculations are to be done, but for the DESS implementation, the state is updated when power information is read from an object as was described above using the `Get_Power_Info_From` method [29:16].

As mentioned previously, each system is represented by a system aggregate. The system aggregate names each object and the names are used to reference an array of pointers to the system's objects [29:23]. This provides named access to any object in a system through the system aggregate. Figure 7 is a fragment from the DC Power System Aggregate package showing how the objects are named and instantiated in a constant array indexed by object name.

The DC Power System connections are defined in a data structure as shown in the code fragment in Figure 8. As noted before, each connection has only two connection points. These connection points are implemented as an array of size two with each element in the array declared as an element point defined in the DC Power System Aggregate package.

The DC Power System package defines procedures to process all of its internal or system-level connections. For voltage and LCF, connections are processed by reading point 1 of a connection and writing to point 2 of a connection in order from `Connection_7` to `Connection_42`. Thus, all connections are processed for voltage and LCF such that voltage flows from left to right (source

```
package Dc_Power_System_Aggregate is

     type Cb_Names is (
     -- CB's between TRUs and tie_bus_1
     --
       Cb_1_1,
       Cb_2_1,
       Cb_3_1,

     -- define a table in which the objects are instantiated and
     -- can be referenced by the name.
     --
     Named_Cbs : constant array (Cb_Names) of Cb_Object_Manager.Cb := (
            Cb_1_1 => Cb_Object_Manager.New_Cb
                 (Position => Cb_Object_Manager.Closed, Rating => 50.0),
            Cb_2_1 => Cb_Object_Manager.New_Cb
                 (Position => Cb_Object_Manager.Closed, Rating => 50.0),
            Cb_3_1 => Cb_Object_Manager.New_Cb
                 (Position => Cb_Object_Manager.Closed, Rating => 50.0),
            ...);
```

Figure 7. DC_Power_System_Aggregate Package Code Fragment [29:23]

to sink) in Figure 3. For load, connections are processed in reverse order (from `Connection_42` to `Connection_7`). Because tie buses have voltage and current flowing in both directions, the connections to tie buses are processed more often than other connections [29:24,25].

Figure 9 depicts the overall software architecture of a typical aircraft simulator. The architecture diagram is displayed in the notation presented by Grady Booch in [7]. The diagram depicts that the **Flight_Executive_Connections**, **Engine_System**, **DC_Power_System** and the **AC_Power_System** packages are all "with"-ed by the body of the **Flight_Executive** package, the **Flight_Executive_Connections** body *withs* the **DC_Power_System_Aggregate**, the **CB_Object_Manager** packages, etc..

"Each system is represented by a package called **{system_name}_System**. The specification of each system package exports a single procedure, **Update_{system_name}_System**, which is called by the flight executive to update the system" [29:30]. Thus, the state of the DC Power System is updated by a call to the procedure **Update_DC_Power_System** that is called from the

```
package body Dc_Power_System is

  package DCPA renames Dc_Power_System_Aggregate;

  type Dc_Power_System_Connection_Names is (
     -- Connections from TRUs to dc_mains 1, 2, and 3
     --
     Connection_7, Connection_8, Connection_9,

     -- Connections from Dc_Main_1 to power bus CBs
     --
     Connection_10, Connection_11,
     ...);

  -- Structure for each Connection
  --
  type A_Dc_Power_System_Connection is array (Integer range 1 .. 2) of
                 DCPA.Element_Point;

  type Dc_Power_System_Connections is array
                 (Dc_Power_System_Connection_Names) of A_Dc_Power_System_Connection;

  -- Table which provides a cross reference between each Connection
  -- and information on each Point on it.
  --
  The_Dc_Power_System_Connections : constant
         Dc_Power_System_Connections := (

            Connection_7 => (
                      2 => (Element => Global_Types.A_Bus,
                           Bus_Element => DCPA.Dc_Main_1,
                           Bus_Side => 1),
                      1 => (Element => Global_Types.A_Tru,
                           Tru_Element => DCPA.Tru_1,
                           Tru_Side => Tru_Object_Manager.Dc_Side)),

            Connection_8 => (
                      2 => (Element => Global_Types.A_Bus,
                           Bus_Element => DCPA.Dc_Main_2,
                           Bus_Side => 1),
                      1 => (Element => Global_Types.A_Tru,
                           Tru_Element => DCPA.Tru_2,
                           Tru_Side => Tru_Object_Manager.Dc_Side)),
              ...);

end Dc_Power_System;
```

Figure 8. DC_Power_System Package Body [29:125,126]

body of the `Flight_Executive` package. The `Flight_Executive` package exports the single procedure `Update_Flight_Executive` that gates all the executive-level connections to a system (via the `Flight_Executive_Connections` package), then it calls `Update_{system_name}_System` for each system that is visible to the flight executive. The updating of the system is executed by using a tabular schedule of systems to update [29:28]. "The names of the systems are declared in the package `Flight_Systems_Names`, the sole purpose of which is to enumerate the names" [29:28].

The executive-level connections are defined in the `Flight_Executive_Connections` package, and this package contains procedures to gate systematically all the executive-level connections in the flight executive. The system-level connections are declared in the `DC_Power_System` package; although, the `DC_Power_System_Connections` package is shown as separate from the `DC_Power_System` package in Figure 9. "The separate package (for `DC_Power_System_Connections`) is drawn for notational simplicity" [29:27]. All the dependencies originate from the bodies of the packages to reduce the need for widespread recompilation if there is a code change in one of the packages.

3.6 Overview Summary

The SEI's OOD Paradigm provides an informal method to map real entities from a description such as a schematic diagram to a graphical design specification. The key abstractions of the design are:

- Objects
- Connections
- Systems
- Executives.

Each of these symbols maps to a component within a software architecture.

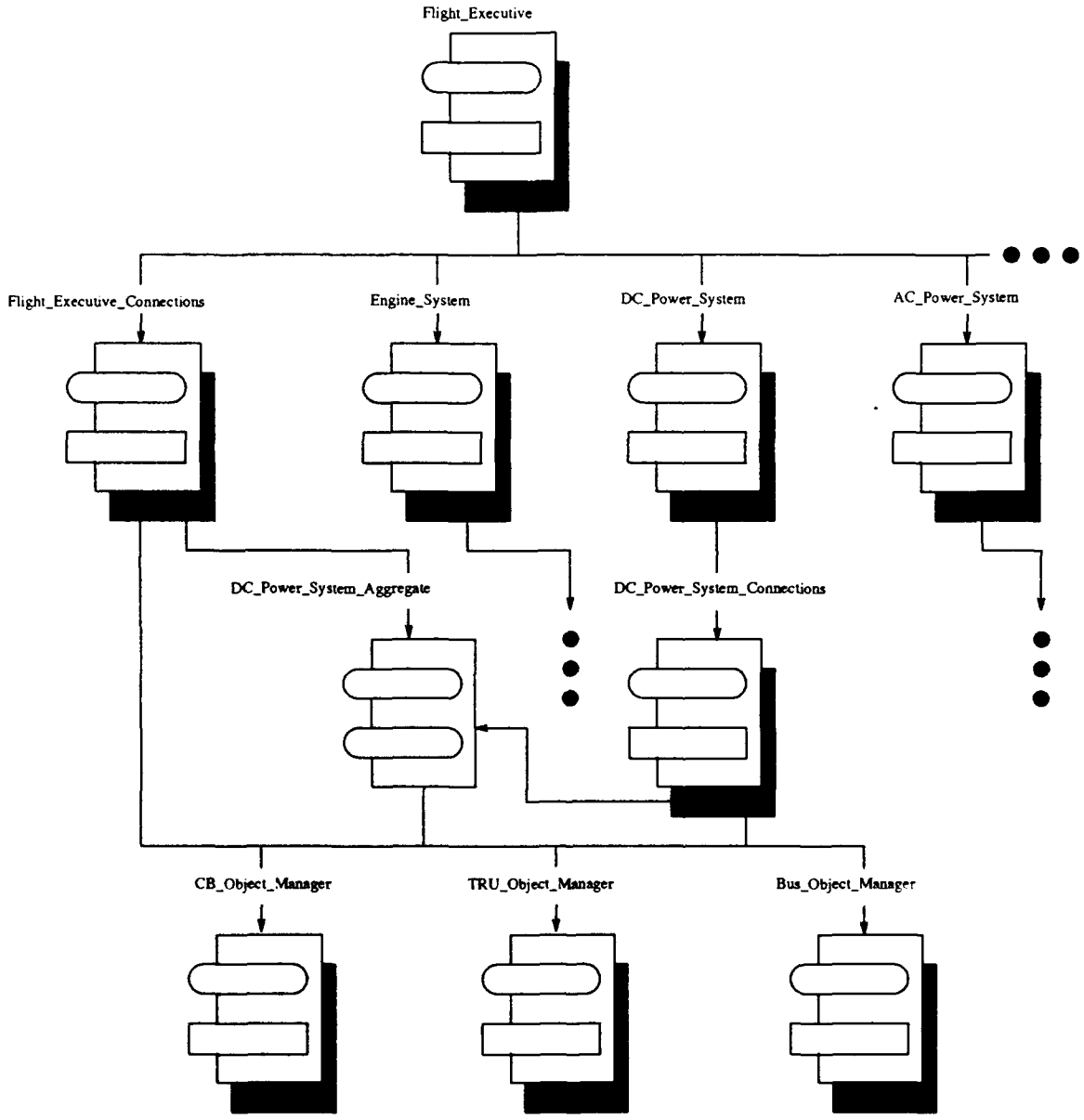


Figure 9. Executive-Level Software Architecture

A class of objects is implemented by an object manager package that defines the methods and procedures for that object. Connection packages are implemented for systems and for the executive. The connection packages for systems are nested within the system package, but the flight executive connections are in the `Flight_Connections` package. Each system is defined by a system aggregate that names all objects within the system. The connection packages have visibility to the aggregate package to have named access to the system's objects.

Executive-level connections to a system are gated prior to gating the connections to a system. Once all system-level connections are gated the system is in a consistent state. Each system provides a single procedure, `Update_{system_name}`, that gates all connections within a system.

The paradigm is designed with the concept of parallel execution of executives on separate processors, but no explicit method is provided to support parallelism. The question of how to extend this paradigm and its associated architecture to support concurrency is addressed in the next chapter.

IV. Analyzing the SEI OOD Paradigm for Concurrency

4.1 Introduction

Concurrency may exist at many levels within a simulation designed using the SEI OOD Paradigm. There may be a course-grain concurrency at the system level, medium-grain concurrency at the object level, or fine-grain concurrency at the instruction level. The speedup that can be achieved in parallelizing a simulation designed by using the SEI OOD Paradigm may be limited by the amount of parallelism that can be used at each of these levels. This chapter presents an analysis of the concurrency that exists within a simulation designed using the SEI paradigm.

The goal of this analysis is to find a level of concurrency that can be implemented for simulations designed using the SEI paradigm. The SEI's DC Electrical System Simulation (DESS) will be the specific example used to conduct this analysis, but the analysis will be done so that it can be applied to other simulations designed using the SEI's paradigm, or to other systems designed using the paradigm, e.g., the engine system, the fuel system, etc..

An informal analysis using English-language descriptions is presented, as opposed to a formal logic analysis using a language such as Chandy and Misra's parallel design language, UNITY [10].[1] The first section of the analysis addresses the factors that affect the potential speedup that can be achieved using any level of concurrency, and the following sections address the potential course-grain, medium-grain and fine-grain concurrency that can be added to the design of a simulation based on the SEI's paradigm.

4.2 Analysis

4.2.1 Factors that Affect Potential Speedup. The goal of implementing concurrency in a simulation (or any other program) is usually to decrease the execution time of the simulation

[1] Using the UNITY language provides a good representation for describing and proving the correctness of a parallel algorithm; however, an English-language description is presented here.

(or other program). This decrease in execution time is called *speedup* and it is defined by the equation [14:4]:

$$S_p = \frac{T_1}{T_p} \quad (1)$$

where T_1 = execution time of the best sequential program on a single processor

T_p = execution time of parallel program using p processors.

To achieve a linear speedup, the value of T_p must decrease in direct proportion to the number of processors used. Ideally, going from one processor to two processors should yield a speedup of $S_2 = 2$, using four processors should yield a speedup of $S_4 = 4$, etc.. This type of linear speedup is desirable, but it may not be achievable due to the effects of load imbalance, communications overhead and synchronization or serial dependencies between processors. Any level of concurrency that can be exploited in the design and implementation of a parallel simulation based on the SEI OOD Paradigm will be affected by these three factors.

4.2.1.1 Load Imbalance. If the total amount of work (load) performed by one processor is equal to W_{seq} with execution time T_1, then for p processors the execution time for each processor should be $\frac{T_1}{p}$ if each processor has a balanced load equal to $\frac{W_{seq}}{p}$ and communications overhead is negligible. If each processor does not have a load equal to $\frac{W_{seq}}{p}$, then the processors have a load imbalance, and the execution time of the simulation is driven by the processor with the largest load. The execution time of the parallel simulation is equal to the longest execution time of any of the p processors. The execution time for p processors that are load balanced is equal to $\frac{T_1}{p}$, and the speedup of the simulation is equal to

$$S_p = \frac{T_1}{\frac{T_1}{p}} = p. \quad (2)$$

If the load is not balanced on each processor, then the speedup is a function of the maximum execution time of all the processor execution times, $T_{p_{max}}$, where $T_{p_{max}} > \frac{T_1}{p}$ due to the load imbalance. The speedup for a load imbalance is then equal to

$$S_p = \frac{T_1}{T_{p_{max}}} < p. \quad (3)$$

Thus, the speedup for the simulation is less than p for a load imbalance, and a linear speedup is not achieved due to the load imbalance.

4.2.1.2 Communications Overhead. Another factor that can prevent achieving a linear speedup is the overhead due to communications between processors. A simulation such as the DESS will require the exchange of information between processors due to state data being transferred between systems or objects in the simulation. Transferring this information requires a finite amount of time, and this time adds to the execution time of the parallel simulation. For a distributed memory system such as the iPSC/2, the amount of time required to transfer information between processors varies depending on the amount of data transferred between systems [28]. The transfer time increases as the message size increases. The time required to transfer data on each processor will increase the value of T_p for each processor. If the communications operations can be overlapped with computations, or the amount of communications can be reduced; then the impact of the communications overhead on T_p can be decreased.

4.2.1.3 Synchronization/Serial Dependencies between Processors. A parallel program executing on several processors may require synchronization between processors due to the algorithm used by the parallel program. For example, if two processors are each running half a parallel simulation (i.e. each processor has one-half of the workload that would be done by a single processor), and processor 1 generates a value required by processor 2 to execute its part of the parallel simulation, then a synchronization dependency exists between the processors. If processor 2 is idle

waiting for the information from processor 1, then the execution time of processor 2, T_{2_2}, increases by the amount of time spent waiting. If the amount of time spent waiting by processor 2 is $T_{waiting_2}$ and the workload on each processor is $\frac{W_{seq}}{2}$, then the execution time of processor 2 is equal to

$$T_{2_2} = \frac{T_1}{2} + T_{waiting_2} \tag{4}$$

and the speedup of the simulation is equal to

$$S_2 = \frac{T_1}{\frac{T_1}{2} + \frac{1}{T_{waiting_2}}} < p \tag{5}$$

where $p = 2$. Thus, a less than linear speedup will be achieved due to the increase in T_p of processor 2 that is caused by the synchronization dependency.

Eliminating synchronization dependencies or reducing the amount of idle time spent between synchronization points will increase the potential for a linear speedup. For the example discussed above, $T_{waiting_2}$ can be reduced by having processor 1 send its data to processor 2 as soon as possible, and having processor 2 execute as much of the workload that it can prior to waiting for processor 1's data to be received. Decreasing the waiting time using this method is limited by the amount of work that processor 1 *must* do prior to sending data to processor 2, and it will be limited by the amount of work that processor 2 *can* do prior to having to wait for the data from processor 1.

4.2.2 Course-Grain Concurrency. A full simulation developed using the SEI paradigm will contain many systems that may exhibit course-grain concurrency. Course-grain concurrency exists if the systems within the simulation (engine system, fuel system, etc.) can execute independently and asynchronously for large periods of time, and the systems spend small periods of time in synchronous operation. The SEI paradigm was designed with parallelism at the system-level in mind [30:11].

This course-grain concurrency can be used to map a system to each processor on a distributed memory architecture such as the iPSC/2 Hypercube. The processors execute the operations of a system asynchronously until information has to be passed between systems. Information must be passed between systems when executive-level connections are gated. Synchronization occurs for each iteration of the simulation when the executive-level connections are gated; but each system can update itself asynchronously when each system gates its system-level connections.

If a simulation with 100 systems is parallelized by mapping a system to each of 100 processors, then the potential speedup of the parallel simulation is equal to 100 if $T_{100} = \frac{T_1}{100}$. This linear speedup of the simulation is not likely to be achieved due to load imbalances on each processor, communications overhead and synchronization required to transfer state information between systems in the simulation.

A load imbalance of the simulation using system-level, course-grain parallelization may occur due to each system having a workload different than $\frac{W_{seq}}{100}$. Some systems may have a workload greater than this, and others may have a workload less than this. The execution time of the simulation will be driven by the system with the largest workload.

Communications overhead will be introduced due to message passing between systems on different processors. Executive-level connections are used within the SEI Paradigm to transfer information between systems, and gating these connections between systems on different processors will require a finite amount of time to transfer data. The overhead due to communications will be a function of the number of messages sent between processors, the size of the messages and the amount of processor time required to send messages. If computations can be overlapped with communications on each processor, then the effects of communications overhead can be reduced.

Another factor that will decrease the potential speedup is synchronization dependencies between the system on each processor. For example, the DC Power System (DPS) in the DESS receives inputs from the AC Power System (APS) and the Dummy System (which simulates the

load on the DC Power System). The APS provides an AC voltage to the TRUs in the DPS when the executive-level connections between these systems are gated. The Dummy System (DS) sends load information to the DPS when the executive-level connections between these two systems are gated. The DPS then updates its system-level connections based on the inputs received from the executive-level connections. After the DPS has updated its state by gating its system-level connections, then a series of executive-level connections that transfer load information from the DPS to the APS would be gated.[2] Also, the DPS would have executive-level connections that send voltage data to the DS. The APS and DS may not be able to update their states until after the DPS has finished gating its system-level connections and sent its new load and voltage data *via* the executive-level connections back to the APS and DS. Therefore, the potential speedup of the simulation may be reduced to a less than linear speedup due to the synchronization dependencies between systems on different processors.

The final factor to consider in using course-grain concurrency is that the potential speedup for a simulation is limited by the number of system within the simulation. For the example, in the 100 system simulation mentioned above the potential speedup is limited to 100 if the effects of load balancing, communication overhead and processor synchronization are negligible. If the effect of these factors is not negligible the actual speedup could be 50% or less of the potential 100 times increase. If a parallel machine with more than 100 processors (such as a 1024 processor hypercube) is available for running the simulation; then using the course-grain level of concurrency, it will not be possible to use the larger system to achieve a potentially higher speedup. A greater speedup may be possible using a lower level of concurrency. For instance, if a lower level of concurrency could be used such that the 1024 processors are used to execute part of the simulation mentioned previously and if only 20% of the potential 1024 times speedup is achieved, then the *achieved* speedup using the lower level of concurrency would be more than twice the *potential* speedup of the course-grain

[2] In the SEI's implementation of the DESS, the APS and the DS are "stubbed-out" and the executive-level connections that would transfer information from the DPS to the APS and DS are not implemented. If a full implementation of the APS and DS were done, then these connections would be required.

simulation using 100 processors, i.e.,

$$S_{1024} = 0.2(1024) = 204.8 > 2 \times 100 = S_{100}.$$

4.2.3 Medium-Grain Concurrency. Medium-grain concurrency can be implemented at the object level in a parallel simulation designed using the SEI OOD Paradigm. Concurrency appears to exist at the level of operations that are done to objects, such as the gating of connections. This lower level of concurrency is classified as medium-grain concurrency since the grain of sequential operations is the execution of a method on an object.

The potential speedup using medium-grain concurrency is a factor of the number of objects or connections that can be processed in parallel. If each of the 100 systems mentioned above has an average of 60 objects and if five to six objects are mapped to each processor, then all 1024 processors of the hypercube mentioned previously could be used to execute operations on objects in parallel. By concurrently gating the connections that are associated with the objects on each processor node, all 1024 processors can execute in parallel. If delays due to communication overhead are minimal and load balancing of the processors is achieved, the potential speedup limit would be a 1024 times increase instead of the 100 times increase possible through system-level parallelism. By using medium-grain concurrency, more processors may be used to try to achieve a higher speedup.

In the DESS simulation the objects do appear to have concurrency; the objects can change their states simultaneously. However, a problem is seen when the algorithm of the simulation is investigated in more detail. The simulation executes by modeling the flow of energy through the objects in the simulation. For example in Figure 3, page 23, voltage and load conversion factor (LCF) data are transferred from TRU_2 to Main_2, then from Main_2 to CB_2.1 and CB_2.2. The voltage and LCF data for all objects in the DESS is transferred in this way. After all the voltage and LCF values are transferred, load data is transferred from CB_2.1 and CB_2.2 to Main_2, and then from Main_2 to TRU_2. Thus, the processing of load information is in the reverse order and

opposite direction of the voltage processing. Load data for all the objects in the DESS is transferred this way.

The voltage and LCF values sent from Main_2 to CB_2.1 and CB_2.2 ($V_{\text{Main\_2}}$ and $LCF_{\text{Main\_2}}$) are calculated by using the input voltage and LCF from TRU_2, i.e.,

$$V_{\text{Main\_2}} = f(V_{\text{TRU\_2}})$$

$$LCF_{\text{Main\_2}} = g(LCF_{\text{TRU\_2}})$$

The load sent from Main_2 to TRU_2 is calculated using the input loads from CB_2.1 and CB_2.2 and the $LCF_{\text{Main\_2}}$ value previously calculated when the voltage and LCF data where sent to Main_2 from TRU_2. The load sent from Main_2 to TRU_2 is represented by the following equation:

$$\begin{aligned} I_{\text{Main\_2}} &= h(I_{\text{CB\_2.1}}, I_{\text{CB\_2.2}}, LCF_{\text{Main\_2}}) \\ &= h(I_{\text{CB\_2.1}}, I_{\text{CB\_2.2}}, g(LCF_{\text{TRU\_2}})) \end{aligned}$$

Due to the way voltage and LCF data are transferred between objects, and the dependency of load data on LCF data (as shown in the equations above), an ordering for the gating of connections for the objects in the system is required. This ordering of the processing of connections is further complicated by the tie buses. As noted in Chapter III, the tie buses are reprocessed because they have current and voltage flowing in both directions. The tie bus voltages are processed after all the other voltage and LCF connections for the objects are completed. Then the load information is processed for all the connections. When this is finished the load information for the tie buses is processed. The processing of the tie bus voltages, LCFs and loads adds more dependencies to the processing order. Thus, a level of concurrency does exist in the transferring of information between objects; however, the amount of concurrency is conditioned upon the dependencies between the

processing order of the connections. The dependencies within the DESS and their effect on the level of parallelism that can be achieved are addressed further in Section 4.3.

The medium-grain concurrency described is a natural level of concurrency that exists due to the way the SEI paradigm implements updating of systems by gating connections. The operation of a system is simulated by the state of a system being changed through the action of gating connections, and gating of connections is the "action" of the simulation.

An object is acted upon by a connection that requests a service from the object, i.e. when a connection is gated the connection requests information from an object, and then gives that information to another object. The two objects have no knowledge of the connection that exists between them; this information is embedded in the system's connection package.

The parallelism that can be implemented in the gating of connections is limited only by the algorithm used in calculating state information within a system. In the ideal case all the connections could be gated concurrently, and in the worst case each connection must be gated sequential one after another. These levels represent the extremes of connection gating concurrency, and the level of concurrency that can be achieved for a particular system simulation is dependent upon the algorithm and the simulation model being used for that system. For the DESS, the limiting factor to having all connections gate concurrently is the method of modeling the system by the flow of energy between objects. The speedup that can be achieved by implementing parallel-connection-gating for the DESS is a function of the dependencies of the connections in the DESS. Section 4.3 is an analysis of the these connection dependencies.

4.2.4 Fine-Grain Concurrency. Fine-grain concurrency is a level of concurrency for which the grain size is each instruction executed, e.g., addition, multiplication, etc.. From research reported in [28], the iPSC/2 Hypercube does not appear to be as well suited for fine-grain concurrency as it is for medium-grain concurrency. Therefore, medium-grain concurrency is the lowest level of concurrency considered in this analysis.

4.3 Connection Dependencies

The level of concurrency that can be derived using the parallel-connection-gating method described above is a function of the connection's processing-order dependencies. For the DESS simulation, the connection processing-order dependencies are due to the simulation of the circuit by the flow of voltage from one side of the circuit to another, and the flow of current through the circuit in the opposite direction.

All the connections in the DESS must be enumerate in order to analyze the connections processing dependencies. In the original SEI simulation, 52 connections are declared for the DC Power System. Sixteen connections are executive-level connections and 36 connections are system-level connections. As mentioned in Chapter III, the connections are processed in one direction for voltage and LCF data, and in the opposite and reverse direction for load data. Also, 19 of the connections are reprocessed for voltage, LCF and load for the tie buses. If a connection is redefined to be a one-way transfer link from a source to a destination, then the DC Power System has 110 system-level connections, and 16 executive-level connections. Table 1 shows how all the original two-way connections and tie bus connections are renumbered as one-way connections. This numbering scheme is used in the connection dependency graph shown in Figure 10. The connection dependency graph is derived from an analysis of the simulation modeled used in the DESS, and the design of the circuit.[3]

Figure 10 shows the connection paths between objects in the DC Power System and their numbering in a graph format for voltage connections, and load connections. The nodes of the graphs correspond to the objects in the DC Power System, and the arcs are the system-level connections for voltage, LCF and load. The numbers of the connection arcs correspond to the connection numberings from Table 1. If the connections are processed in order from 7 to 116, then all the dependencies of the connection gating are handled correctly (as done in the original

[3] Deriving this dependency graph is very application specific, but may be able to be automated for other applications. For this analysis the graph was manually derived.

Table 1. New DC Power System Connection Enumerations

Original Connection	Type Data Sent	New Connection	Type Data Sent
Connection_7	Voltage, LCF	7	Voltage, LCF
Connection_8	Voltage, LCF	8	Voltage, LCF
...	Voltage, LCF	...	Voltage, LCF
Connection_41	Voltage, LCF	41	Voltage, LCF
Connection_42	Voltage, LCF	42	Voltage, LCF
Connection_23	Voltage, LCF	43	Voltage, LCF
Connection_22	Voltage, LCF	44	Voltage, LCF
...	Voltage, LCF	...	Voltage, LCF
Connection_11	Voltage, LCF	55	Voltage, LCF
Connection_10	Voltage, LCF	56	Voltage, LCF
Connection_28	Voltage, LCF	57	Voltage, LCF
...	Voltage, LCF	...	Voltage, LCF
Connection_24	Voltage, LCF	61	Voltage, LCF
Connection_42	Load	62	Load
Connection_41	Load	63	Load
...	Load	...	Load
Connection_8	Load	96	Load
Connection_7	Load	97	Load
Connection_10	Load	98	Load
Connection_11	Load	99	Load
...	Load	...	Load
Connection_22	Load	110	Load
Connection_23	Load	111	Load
Connection_24	Load	112	Load
...	Load	...	Load
Connection_28	Load	116	Load

sequential program). The graph in Figure 10 shows that several objects have four connections between objects, i.e., connections 23, 43, 81, and 111 for TB 1 (Tie Bus 1) and CB T1.2 (Tie Bus Circuit Breaker 1.2). These connections show that four messages are passed between TB 1 and CB T1.2. The first two messages are voltage and LCF data, and the next two messages are load data.

A directed, acyclic graph of the DC Power System processing ordering connection dependencies is shown in Figures 11 and 12. Each node in the graph represents a connection, and the arcs show the dependencies between the connections. For example, connection 7 is not dependent on any other connection, but connection 7 must be processed before connections 10, 11, and 16 can be gated.

To determine an estimate of the potential speedup that can be achieved using the parallel-connection-gating, the critical path through the connection dependency graph must be found. This critical path is only an indicator of the potential speedup that can be achieved, but if the estimate is less than one, then no potential speedup exists for the DESS using the parallel-connection-gating method due to the serial execution dependencies of the connections in the DESS. If the length of the critical path is greater than one, then the potential for speeding up the DESS exists, but the actual speedup will be contingent upon how efficiently the parallel version of the DESS is designed. The design of the DESS is addressed in the next chapter.

If the time to gate a connection is approximately the same for any connection being processed and the cost of gating a connection is equal to unity, then there are several critical paths in the connection dependency graph that have the same maximum length – 20 units. One critical path that has an execution cost of 20 units is the path: 7, 16, 19, 23, 27, 28, 29, 30, 40, 64, 74, 75, 76, 77, 81, 83, 86, 106, 109, 110. Other paths of length twenty start from 8 and 9 and end at 110 or 111. Since the maximum critical path length is 20, this is the minimum execution time that may be achieved by parallelizing the connection processing. If all the DC Power System connections are processed sequentially then the execution time would be 110 time units (assuming unit execution

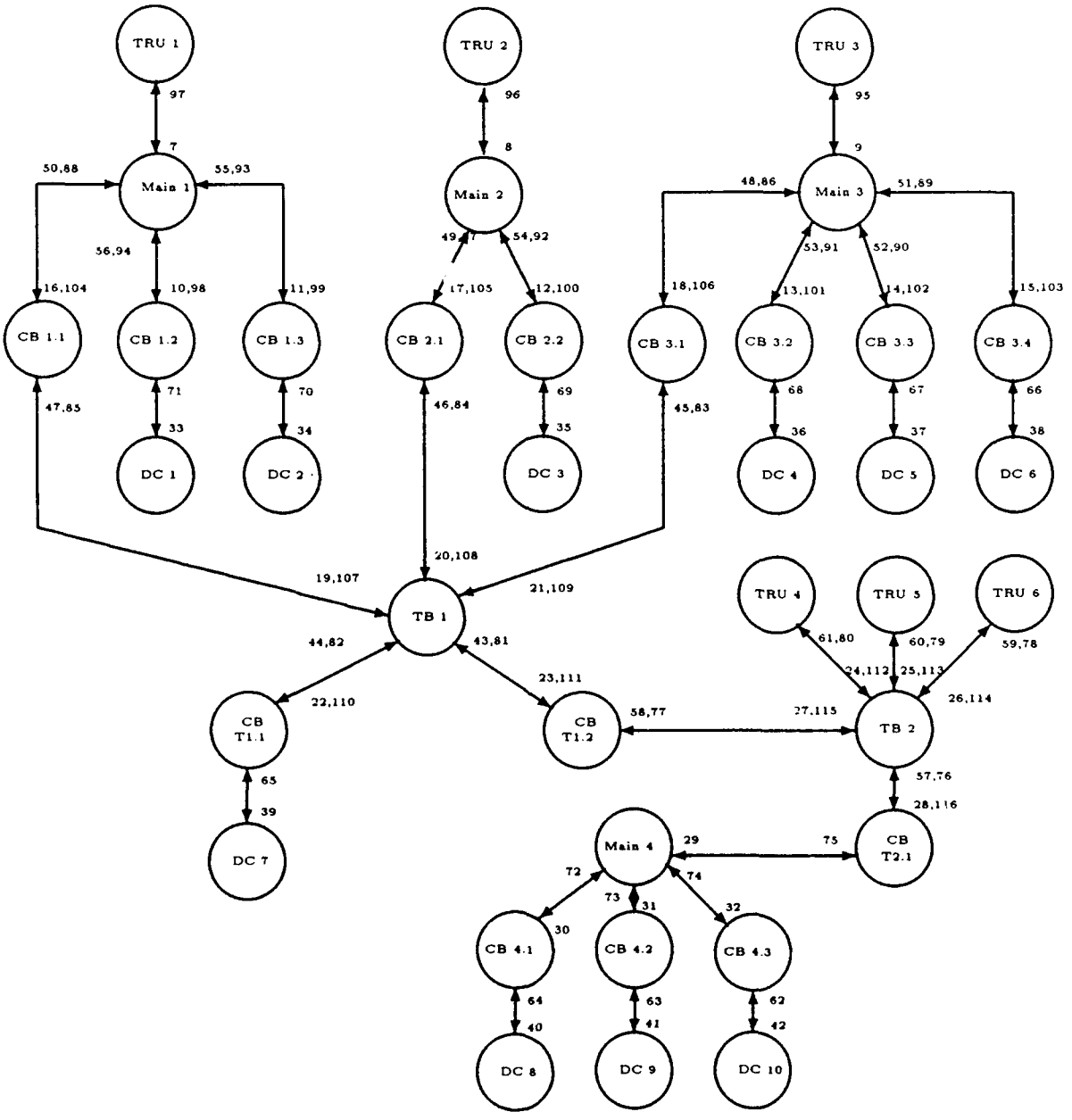


Figure 10. Voltage, LCF and Load Connections Graph

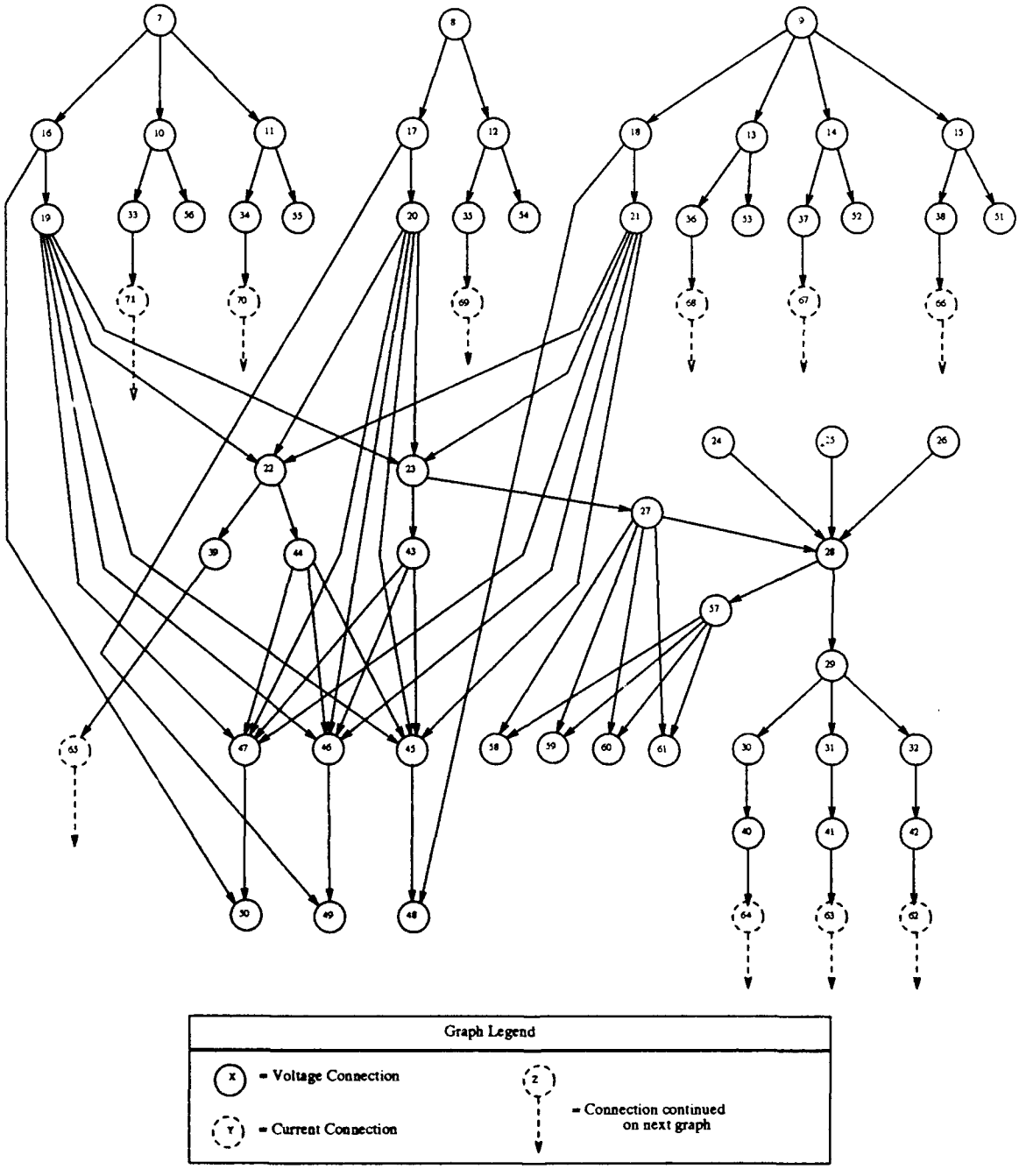


Figure 11. Connections Dependency Graph - Part 1 (Voltage and LCF)

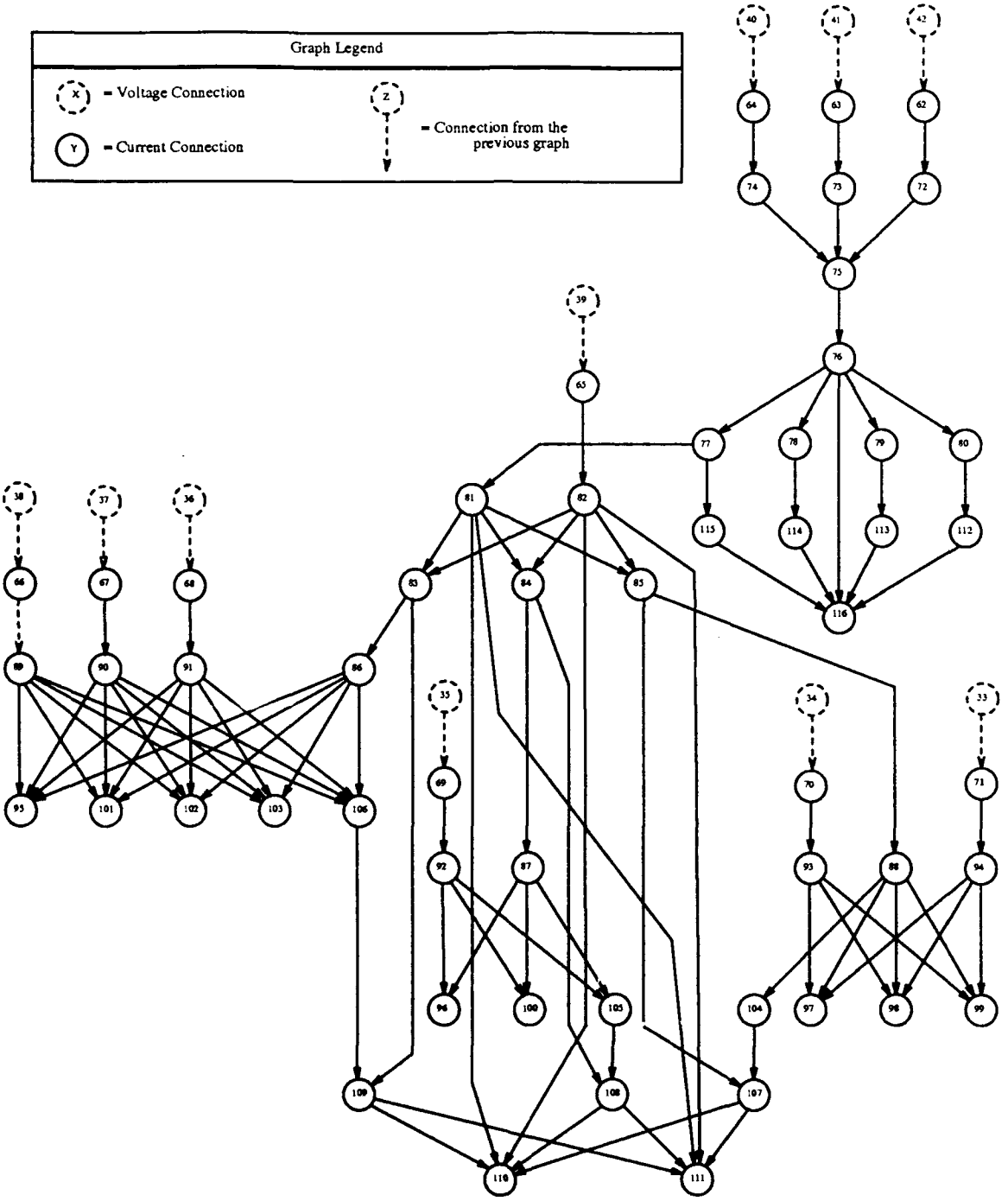


Figure 12. Connections Dependency Graph - Part 2 (Load)

time for each connection processed). Thus, the upper limit on speedup for the DC Power System is

$$\frac{time_{sequential}}{time_{criticalpath}} = \frac{110}{20} = 5.5$$

To achieve this 5.5 time speedup at least six processors are needed, and using more than six processors will not result in any further speedup. To obtain the 5.5 time speedup potential the connections must be able to be scheduled on the six processors so that all the connections are processed in order and no processor requires more than 20 time units to execute. One schedule that meets this requirement is shown in Table 2.

The schedule in Table 2 was generated by assuming that the cost of processing each connection was the same and equal to unity. If processing one connection involves reading state information from one object on a node and sending that information to an object on another node, the assumption of unity time for connection processing may not be correct due to the time that it takes to send a message between nodes. For the iPSC/2 Hypercube the average time per byte of sending a message is as shown in Table 3. The average time to execute a floating point operation is shown in Table 4. Table 3 shows that the transfer time per byte decreases as a function of the message size sent. If we compare the time that it takes to send a floating point value against the average time that it takes to do a floating point calculation (0.00883 msec), then we get an idea of the cost of sending a floating point value in terms of the number of computations that can be accomplished while the message is being sent. Using values from the tables, we see that in the time it takes to transfer one floating point number (8 bytes) between nodes, 44.5 floating point operations (FPOs) can be executed. If 128 bytes (16 floating point values) are sent between two nodes then 5.1 FPOs can be done in the time that it takes to transfer one floating point value. Although the total transfer time for the 128 bytes is longer than the time it takes to transfer 8 bytes, the number of FPOs that

Table 2. Six Processor Connection Processing Schedule

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
P1:	7	10	13	22	27	28	29	30	40	61	67	70	76	77	81	83	86	95	98	101
P2:	8	11	14	23	37	45	48	31	41	62	68	71	91	78	112	84	87	96	99	102
P3:	9	12	15	33	38	46	49	32	42	63	69	75	92	79	113	85	88	97	100	103
P4:	24	16	19	34	39	47	50	55	58	64	72	82	93	80	114	116	–	104	107	110
P5:	25	17	20	35	43	51	53	56	59	65	73	89	94	–	115	–	–	105	108	111
P6:	26	18	21	36	44	52	54	57	60	66	74	90	–	–	–	–	–	106	109	–

Table 3. Measured iPSC/2 Communication Times (msec/byte)

Block Size (bytes)	Time for Block (msecs)	Avg. Time per byte (msecs/byte)
1	0.3438	0.3438
8	0.3930	0.04913
128	0.7213	0.005635
512	0.8581	0.001676
1024	1.0450	0.001021
2048	1.4125	0.0006897
4096	2.1375	0.0005219

Table 4. Average Floating Point Computation Time (msec)

Addition	Subtraction	Multiplication	Division	Average
0.00740	0.00770	0.00870	0.01150	0.00883

can be done per transfer of a floating point value decreases for a 128 byte message. This shows that the cost of processing a connection that transfers data between nodes may be more than the cost of executing a connection that does not transfer data between nodes, and this cost is a variable cost that is function of the size of the message sent when processing a particular connection. A message consisting of voltage and LCF data would be approximately 10 bytes in size (8 bytes for LCF floating point type, 1 byte for the voltage discrete type, and 1 byte overhead for the record that contains these values). A load message would be 8 bytes – the size of a single floating point value. Thus, approximately 40 FPOs can be done in the time that it takes to send voltage and LCF, or load data between two iPSC/2 nodes. If the destination processor has 40 FPOs to execute before it needs the data sent by the source processor, then the cost of the communications will not be noticed since the communications will be overlapped by computations. However, if the destination processor has fewer than 40 FPOs to execute, then the processor will be delayed by the amount of time that it must wait to receive the incoming data. Therefore, calculating the cost of processing a connection that transfers data between two processor nodes is contingent upon several factors and may have a cost greater than unity.

The above discussion shows that minimizing the cost of processing connections between processors is an important consideration in implementing the connection-level concurrency, and the design should be implemented so that the cost of processing connections is minimized by overlapping communications and computations. The potential speedup that can be achieved is limited by the costs associated with the processing of connections between processors.

4.4 Analysis Summary

Two levels of concurrency may be implemented for a simulation built using the SEI OOD Paradigm and run on the iPSC/2 Hypercube. Course-grain system-level parallelism is one level of concurrency that can be implemented. This level of concurrency is implemented by mapping systems to processors and each system running asynchronously to update itself, and synchronizing when executive-level connections are processed between systems. The potential speedup that can be gained using system parallelism is limited by the number of systems in the simulation, the communication costs and the synchronization dependencies between systems.

The other level of concurrency that can be implement is medium-grain concurrency done at the object level. Several system-level connections can be gated in parallel if all processing order dependencies are handled correctly. Due to processing order dependencies in the DESS, a potential 5.5 time speedup is possible if communications overhead is minimized. This 5.5 time speedup would add to any speedup that can be gained by implementing course-grain system parallelism. To achieve the maximum speedup possible, the parallel gating of connections must be done efficiently by the parallel processing algorithm. The next chapter addresses the design of the parallel processing mechanism needed to implement course-grain and medium-grain concurrent connection processing.

V. Design and Implementation of Parallel Extensions

5.1 Introduction

The SEI Paradigm exhibits two levels of concurrency that can be used to design and implement a parallel simulation. Large-grain concurrency can be implemented by having multiple systems within the simulation running on separate processors. Medium-grain concurrency can be implemented by gating connections in parallel within each system. The large-grain concurrency and the medium-grain concurrency can both be implemented by designing a connection-processing mechanism that handles executive-level connections and system-level connections in parallel. The following section presents the design of a parallel connection-processing mechanism using these concepts.

5.2 Parallel Design - High-Level

5.2.1 Adding Parallel Communications. The SEI Paradigm has four primary components that comprise any design: objects, connections, systems, and executives. To implement concurrency within the simulation, one or more of these components must handle the communications between objects on separate processor nodes. In extending the design of these objects to implement concurrency, the original design goals of the SEI OOD Paradigm must be considered, and the parallel design extensions should not significantly modify the architecture of the original design so that the benefits of the paradigm's design architecture are maintained. Significant changes to the paradigm's design architecture may reduce the benefits of reusablity provided by the design [41], or may make the parallel architecture too application specific to be applied to other simulations designed using the SEI Paradigm. The method used to implement the concurrency in the simulation should be done without invalidating the design goals achieved in the original sequential design.

A goal of the SEI paradigm is to simplify dependencies between objects [30:8]. The objects, as implemented in the original DESS simulation, only depend on the `Global_Types` package. The

Cb, TRU and bus objects have no dependencies on each other, and the objects have no knowledge of connections between each other. Having objects handle communications between each other on different processors will introduce dependencies between objects and violate one of the design goals of the paradigm because objects will need to have knowledge of their connections to each other. Thus, objects are not the components that should be used to implement the parallel communications between the objects on different processors.

Connections were redefined in Chapter IV as one-way transfer links between objects. The state of a simulation is changed by gating connections between objects based on an order of processing defined by the method that the simulation uses to calculate the state of objects. Gating a connection involves reading the state of a particular variable (or attribute) of one object and applying that state information to another object. The object that the state is read from is the connection's source and the object that the state is written to is the connection's destination. As part of processing a connection, the state information from the source can be converted into the correct type for the destination object. The conversion of data by a connection is allowed as part of the connection processing to help eliminate dependencies between objects [30]. Because connections transfer information between objects, connections must know the locations of objects to transfer information between objects properly. Thus, connections are a logical place to add the design extensions needed for parallel communications between objects on separate processors.

5.2.2 Connection Gating. Systems and executives do not have to deal with parallel communications between processors if the connection packages handle communications between objects. The system-level connections package can provide a connection gating procedure that gates local intra-system connections between objects on the same processor and gates non-local intra-system connections between objects on different nodes. This gating procedure provides the same functionality seen in the sequential version of the simulation, and provides a single method for gating any system-level connection. Executive-level connections require a gating procedure almost identi-

cal with the gating procedure for system connections, except that the executive connection gating procedure must gate local and non-local connections between systems (inter-system connections).

The following algorithm defines the operations of the connection gating procedure:

1. Check to see if the connection source is local.

   - If the connection source is local, then execute a local **get_{attribute}** on the source object.
   - If the connection source is not local, then execute a non-local **get_{attribute}** on the source object. This involves reading a message from an input buffer, or requesting information from a remote object. (The actual method to use for retrieving data from a remote object is a design decision addressed in the Section 5.2.2.1).

2. Convert, if necessary, any of the input data before sending the data to the destination.

3. Check to see if the connection destination is local.

   - If the connection destination is local, then execute a local **put_{attribute}** on the destination object.
   - If the connection destination is not local, then execute a non-local **put_{attribute}** for the destination object. This involves sending a message to another processor, or requesting a remote operation on an object.

This gating method has a time complexity that is a function of the time required to execute a local or non-local **get_{attribute}**, the time required to execute any transformation of the data received from the **get_{attribute}**, and the time required to execute a **put_{attribute}** on an object. The time complexity of the **get_{attribute}** method may vary depending on the type of object to which the method is applied. For bus objects, the time complexity of the **get_{attribute}** is a function of the number of sides that a bus has, but the number of sides that a bus has is constant.

Therefore, the **get_{attribute}** for buses has a time complexity of $O(constant)$, and the time complexity of the gating procedure is constant time, $O(constant)$.

Objects within the simulation maintain internal state variables that are changed by gating connections. These state variables are changed based upon inputs applied to the sides of the objects. For objects such as buses that have several inputs and outputs, all inputs must be received before the outputs are calculated. Thus, before a connection can be gated, all the inputs to the source object must be applied by either another connection being gated or the object's inputs being initialized. If the DESS connections are processed such that all the dependencies between connections shown in Figures 11 and 12 are met, then all the required inputs for each object will be received prior to the object's state being read and sent to another object.

5.2.2.1 Retrieving Data and Sending Data to Non-Local Objects. Only one instance of an object is generated in the sequential simulation, and that single object maintains the state information for the object. In the sequential simulation, when a connection is gated a single source object is read and a single destination object is written.

Gating connections for the parallel simulation involves retrieving information from objects located on different processors. Two possible methods of handling the gating of connections with source objects and destination objects on separate processors are: to have multiple copies of objects on a processor or to have a single copy of any given object on a single processor.

- *Multiple Copies of Objects.* In the parallel simulation it is possible to have multiple copies of an object on different processors, e.g., CB 1.2 could be instantiated on processor 1 and processor 2 with each processor maintaining a separate copy of the state of CB 1.2. The advantage to having multiple copies of an object on a processor is that all the objects needed to gate a series of connections can be local copies of objects. This keeps the gating of connections to local communications only, and thus reduces the cost of gating each connection on a processor. In the connection gating algorithm described above, this would mean that only the local

get_{attribute} and **put_{attribute}** procedures are executed when a connection is gated. Thus, by having a mechanism that keeps the states of the CB 1.2 on processor 1 consistent with the copy of CB 1.2 on processor 2, only local get and puts are needed to gate connections and the cost of gating a connection is minimized.

The disadvantages of multiple copies of an object are, 1) that the mechanism required to maintain consistency between two, three, four or more copies of an object on separate processors is complex, and 2) message traffic between copies of objects on different processors will be high if the state of an object changes often. The multiple copy method is complex because passing messages between object copies and making sure the correct state is consistently maintained requires a way of determining which incoming state message is the most current. Also, the message traffic introduced in passing state information between object copies is high if an object has several state transitions due to several **puts** and **gets** being executed on the object by several different connections.

- *Single Copies of Objects.* It may be simpler and more efficient to have only one copy of an object on any given processor, e.g., there is only one copy of CB 1.2 on processor 1. If a connection that has CB 1.2 as its destination is gated on processor 2, then a message is sent from processor 1 to processor 2 with the data from the connection source object on processor 1.

One advantage of having a single instance of an object on only one processor is that the object's state information is located in only one place. Thus, since the state of an object is maintained in only one location, the consistency problem is eliminated. When a connection is processed for objects on different processors only a single message is sent between objects, and the potential message traffic between processors is reduced.

Another advantage of single copies of objects on processors is that less memory is required than when multiple copies of objects are spread across several processors in the

simulation. The rest of this design is based on having only one copy of an object instantiated on any processor in the parallel simulation.

5.2.2.2 Instantiating and Maintaining Location Information of Objects. The system aggregates instantiate objects and allow named access to objects, so the system aggregates are the place to add extensions for determining where an object is. Each system aggregate can maintain a system object map that has the location of all the objects that are part of that system. The system aggregate can instantiate the objects located on that processor node, and provide the location of other system objects on other nodes. When a connection is gated, the connection gating procedure in the connection package can query the system aggregate package to determine the location of the connection source and destination. Using the system aggregates to maintain location information for objects in the system fits within the functions defined for the system aggregates in the SEI's paradigm, and does not require significant changes to the way system aggregates are implemented in the sequential simulation.

The DESS has three systems, the AC Power System, the Dummy System and the DC Power System. A copy of the system aggregates for each of these systems is on each processing node in the parallel DESS simulation. Each aggregate can read information from an object map that identifies where each system's objects are located. Each aggregate instantiates only the objects that are mapped to that processor. When executive-level connections are gated, the flight executive connections package will query the AC Power System Aggregate, the Dummy System Aggregate and the DC Power System Aggregate to determine the location of the source and destination objects for the executive-level connection being gated, and local and non-local inter-system get and put procedures will be executed based on the relative locations of system objects. When a system-level DC Power System connection is processed, then the DC Power System Aggregate is queried to determine the location of the source and destination objects when the system-level gating algorithm is executed. Local and non-local intra-system puts and gets are executed based on where the source

and destination objects are located. The executive-level connection processing is executed by calling the `Update_Flight_Executive` procedure as is done in the sequential version of the simulation. The system-level connections are gated by executing the `Update_{system_name}_System` procedure. For the DC Power System, this procedure is the `Update_DC_Power_System` procedure.

5.2.3 Connection Processing Dependencies. Connection processing dependencies must be handled correctly if the parallel simulation is to generate the same object states that the sequential simulation generates. An efficient method of connection processing must be done, and the method must allow parallelism in the execution of connection gating. Also, the method should be easily scaled for various numbers of processors. The connection processing mechanism can use several approaches to determine the order in which to gate connections.

One method of connection process ordering is to have a topological list of connections that need to be processed on each node. The list would be a list of connections that need to be processed for the objects that are located on a particular processor. The topological list is ordered such that if the first connection is processed, and then the second connection is processed, etc., then the connection dependencies are handled such that all the dependencies are met. Horowitz and Sahni [25] describe various algorithms for generating a topological list using a topological sorting algorithm. The algorithm they present can be used to generate different topological lists based on the dependencies between nodes in a graph such as the one shown in Figures 11 and 12.

There are several possible topological orderings for connection processing for the connection dependency graph shown in Figures 11 and 12. The topological list method uses only one of the many possible orderings. The number sequencing of the DESS system-level connections shown in Table 1 is a topological ordering of connections 7 through 116 such that if each connection is processed in order from 7 to 116, then all the connection dependencies shown in Figures 11 and 12 are met. This topological ordering was generated by manually applying Horowitz and Sahni's

Topological Sorting Algorithm [25:303], and is also based on selecting an ordering that has the same sequence of connection gating as in the sequential simulation.

The topologically-ordered list is maintained in an array data structure and the connections are gated by gating the first connection in the array list, then the second connection, etc.. When using the topological list method, if a connection is not ready to be gated, then the processor blocks until the connection is ready to be gated. Therefore, the topological list method guarantees that all dependencies are properly handled since all connections are only processed in the order of the topological list.

Another method of determining which connection to gate is to use a graph traversal algorithm that identifies which connections can be gated based on the dependency graph and the connections that have been previously gated. This graph method may allow a higher level of parallelism than the linear list method by allowing a determination to be made about what other connections can be processed if the chosen connection cannot be processed.

The following example illustrates how the topological list method and the graph method work for a simulation that has eleven objects and eleven connections, as shown in Figure 13(a). The object diagram shown in Figure 13(a) shows that object A sends its voltage, V_A, to object B when connection 1 is gated. When connection 2, 3 and 4 are gated V_B is sent to objects C, D and E, etc.. The dependency graph shown in Figure 13 (b) depicts the processing dependencies between the connections of objects A through K. (These dependencies are similar to the dependencies between objects in the DESS, but do not show additional dependencies that occur if load data and tie bus connections are shown.) If objects A, B, C, F and I are mapped to processor 1, and objects D, E, G, H, J and K are mapped to processor 2 (as shown in Figure 14), then some parallelism in connection processing can be achieved. Connection 1 can be gated by processor 1 first, then connections 2, 3 and 4 can be gated. The gating of connection 3 and 4 by processor 1 requires a message, V_B, to be sent to processor 2. When processor 2 gates connection 3 and 4, it reads the V_B messages

sent from processor 1 and applies V_B to objects D and E to complete the gating of connections 3 and 4. Then processor 2 can gate connections 7 and 8. After connection 8 is processed, processor 2 can gate connection 11; but connection 10 cannot be gated until V_C is received from processor 1. After sending the messages for connections 3 and 4, processor 1 gates connection 5 and 6, then connection 9. Processor 1 is now finished processing connections for this iteration of the simulation. After gating connection 6, a message was sent to processor 2 with V_C; and upon receiving V_C from processor 1, processor 2 gates connection 6 by applying V_C to object G, and then gates connection 10. This completes the processing of the connections by processor 2, and the objects on both processors are now in a consistent, updated state.

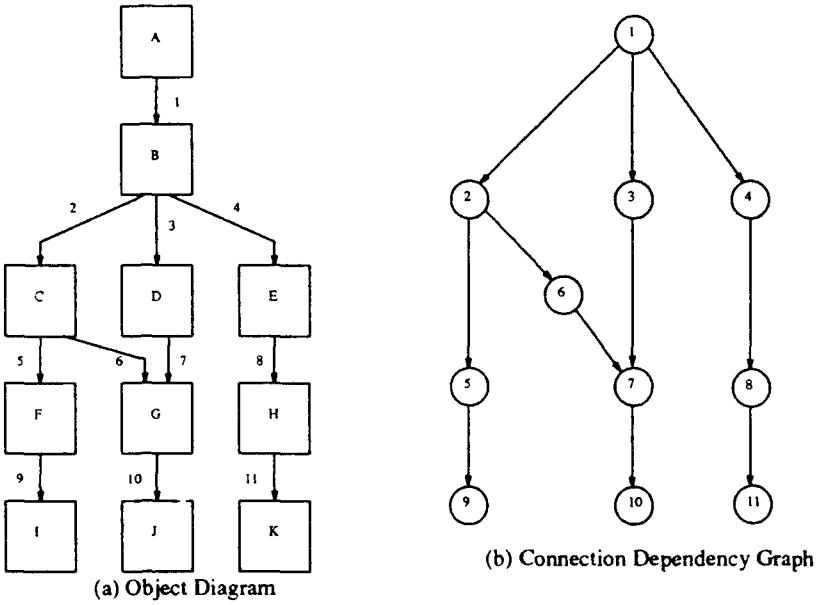


Figure 13. Example Object Diagram and Connection Dependency Graph

If processor 1 gates the connections in a topological order of {1, 2, 3, 4, 5, 6, 9} and processor 2 gates connections {3, 4, 6, 7, 8, 10, 11}; then each processor will correctly process its connections as long as processor 2 waits to gate connections 7 and 8 until V_B is received from processor 1, and connection 10 is gated after receiving V_C. Processor 2 waiting to receive V_B and V_C represents a point of synchronization between processors 1 and 2. This synchronization can be accomplished by using a blocking receive on processor 2. Processor 2 can attempt to gate connection 3 by

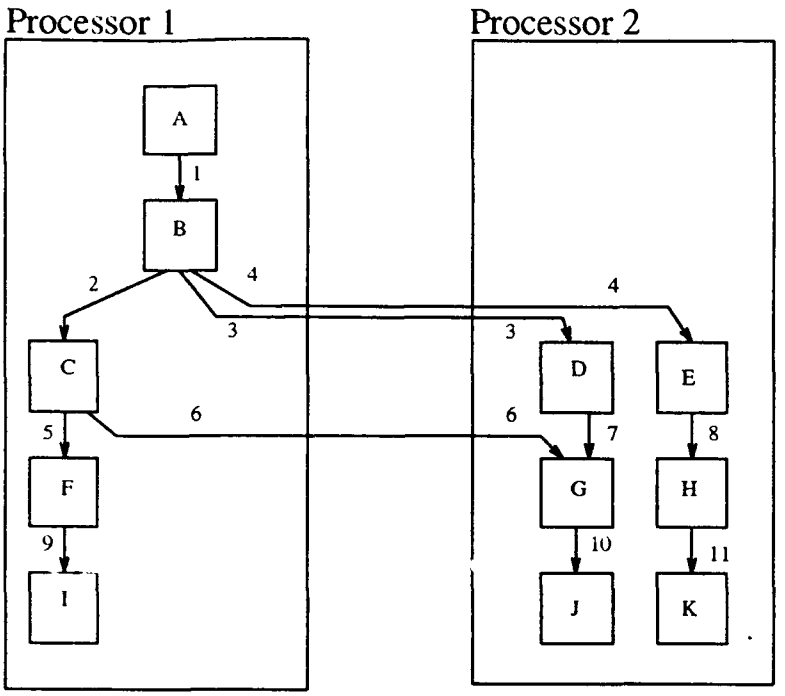


Figure 14. Example Object to Processor Mapping

checking to see if V_B has been received from processor 1. If V_B has been received, connection 3 is gated by reading V_B from the processor's input buffer and applying V_B to object D. If V_B is not available, then processor 2 can block until V_B is received for connection 3. After V_B is received by processor 2, connection 4 would be gated similarly. Since processor 2 cannot execute any other connections until V_B is received, there is nothing else that processor 2 can do except wait until V_B is received. By blocking until each message from processor 1 is received, then all processing dependencies are guaranteed to be handled properly because the topological order of connection processing is maintained. All the connections will be processed on processor 2 in the order {3, 4, 6, 7, 8, 10, 11}.

Some inefficiency is introduced by using the topological list method if processor 2 blocks to receive a connection, such as connection 6, when it could be processing connection 8 and then 11. Connection 8 can be processed anytime after V_B is received and applied to object E, connection 4's destination object. So if processor 2 is blocking to receive connection 6, then connection 8 could be

processed, and connection 11 processed after that. By blocking for connection 6 without considering the possibility of processing connection 8, processor 2 is idle when it could be executing other work. This inefficiency can be reduced by having a mechanism that would execute connections 8 and then 11 while V_C was being sent from processor 1 to processor 2. This mechanism (called the graph method above) must be able to determine that connection 6 cannot be processed, and determine that connection 8 and then 11 can be processed. This requires that the graph dependencies be analyzed to determine what other connections can be gated instead of connection 6, and the mechanism must have the capability to process connection 6 when V_C is received by processor 2. This mechanism will need to execute an algorithm similar to the following:

1. Determine the next connection to be processed based on the connections that have previously been processed, i.e., if connection 1, 2, 3 and 4 have been processed already, then connections 6, 7, and 8 are possible connections that can be processed.
2. Choose one connection to be processed, i.e., connection 6. Determine if that connection can be processed, or if the connection will be blocked because the required input is not yet available from another processor. For connection 6, if V_C was not available then connection 6 cannot be processed.
3. If the connection chosen is not able to be gated (connection 6), then choose another connection (such as connection 7) and determine if it can be gated. Since connection 7 is a local connection, and since connection 3 has been gated, connection 7 can be gated.
4. Gate the connection. For our example, connection 7 would be gated and marked as having been gated for this iteration.
5. Go back to item 1, and determine which ungated connection can be gated.

This algorithm requires that the dependencies of the connections be determined, and that the connections that receive input from other processors be probed. Analyzing the connection depen-

dency graph for the next available connection involves more overhead than just going to the next connection in a topological list.

The time complexity of this graph method is a function of the number of connections that are gated and the number of times that the graph must be searched for a new connection that can be gated without blocking. If there are C connections to gate, and none of the connections are blocked due to unavailable data, then the time complexity of the algorithm is $O(constant)$ since the number of connections to be gated is a constant. However, if connections do block, and a search of the connection dependency graph is done to find another connection to gate, then the time complexity has a maximum of $O(C * N)$, where N is the number of connections that can block due to synchronization dependencies. Thus, in the best case the graph method has the same time complexity that the topological list method does, but in the worst case the graph method has higher time complexity. If the time required to execute the operations associated with the graph method is less than the time that the topological list method spends blocking, when it could be executing productive work, then the additional overhead of the graph method will be beneficial. However, if the additional overhead of the graph method is more than the time spent blocking in the topological list method then no benefit is gained by using the graph method.

Another factor to consider in comparing the topological list method and the graph method is that while processor 2 is blocking to receive V_B, processor 1 may be sending V_C. Then when processor 2 processes connections 3 and 4 after receiving V_B, V_C may arrive and be available when connection 6 is processed. Then the extra overhead involved in doing the graph method may not provide any benefit because processor 6 may not have to block for V_C at all.

A final consideration in using the topological list method and the graph method is potential dead-lock. Dead-lock can occur when two processors attempt to gate connections in such a way that a *circular wait* condition occurs [15:157]. A circular wait may occur if, for example, two processors (processor 1 and 2) gate connections X and Y such that each processor blocks for a message it will

never receive. If processor 1 gates connection X that requires an input from processor 2's connection X, and processor 2 gates connection Y that requires an input from processor 1's connection Y; then, a circular wait will occur if processor 1 blocks for connection X before gating connection Y (which sends data to processor 2) and processor 2 blocks for connection Y before gating connection X (which sends data to processor 1). Each processor will wait for a message that it will never receive due to the circular wait, and a dead-lock occurs.

The graph method avoids dead-lock because eventually processor 1 or 2 will gate connection Y or X, respectively, and the circular wait condition will be broken. These connections will eventually be gated since the graph method searches for connections that can be gated if an attempt to gate a connection is blocked. The topological list method avoids deadlock if each processor maintains the same topological order for the non-local connections that it gates, e.g., processor 1 will gate connection X then connection Y and processor 2 will gate connection X then connection Y. The same topological order can be maintained on each processor if a single topological list is generated and provided to each processor, and each processor only gates the connections in the topological list for the objects that are mapped to that processor.

The graph method adds extra complexity and overhead but may provide only limited increase in performance if an efficient topological ordering is used that minimizes blocking. Therefore, the topological list method will be used for the design of the connection processing mechanism. By having a single topological list on each processor, a good level of parallelism may be able to be achieved and implemented in an efficient way. Also, by choosing a topological ordering that allows messages to be sent to other processors as soon as possible and that allows as much processing and communication overlap as possible to be done, then the topological list method should prove efficient. For the example presented above a good topological ordering for connections on processor 1 is $\{1, 3, 4, 2, 6, 5, 9\}$ and for processor 2 a good ordering is $\{3, 4, 7, 8, 11, 6, 10\}$.[1] These orderings

[1] Note that the topological order of the non-local connections is the same for each processor, e.g., connections {3, 4, 6}.

should minimize delays due to communications and reduce the amount of time that processor 2 spends blocking when it can be gating connections that are not blocked.

5.3 Design Summary - High-Level

The design extensions that are required to add concurrency to the SEI OOD Paradigm are:

1. Provide a single method for the gating of system-level connections and executive-level connections. This method will handle the parallel communications between objects by using the following algorithm,

```
procedure Gate(This_Connection) is

     if This_Connection.Source is local then
          Input_Data := get_{attribute} from This_Connection.Source.Object
     else
          Input_Data := blocking_receive_{attribute_message} from the input buffer
     end if;

     if This_Connection.Destination is local then
          put_{attribute} to This_Connection.Destination.Object
     else
          destination_node = Node_Location_Of(This_Connection.Source)
          send_{attribute_message} to This_Connection.Source.Object on destination_node
     end if;

end Gate;
```

2. Modify the system aggregates so they can instantiate objects on a processor based on an object map provided by the user. The system aggregate still provides named access to objects that are locally available on a processor, but it also provides a method for determining the location of any object in the system based upon the location specified in the object map. Each processor has a copy of each system aggregate needed for objects mapped to that processor. The Circuit Breaker, TRU and Bus Object Managers are the same design as used in the sequential simulation, except the method `Delete_{object}` has been added.

Figure 15 shows the new architecture of the parallel DESS (PDESS). The only change to this view of the design is that the system aggregates now have bodies, but they only had specifications in the sequential version. The next section on low-level design describes the internal changes to packages required to implement the new architecture on the iPSC/2 Hypercube.

5.4 Parallel Design - Low-Level

This section first describes changes required in the global types package and the electrical units package, then the low-level design changes that are required in the package specifications and bodies of each unit of the parallel software architecture shown in Figure 15 are described. The low-level design of the object managers is presented after the changes to the electrical units and global types, then the systems, the system aggregates and the system connection packages are described. The flight executive and the flight executive connections package are described after the system packages. The low-level design of the main procedure packages that run the simulation on the iPSC/2 host and nodes are the last two low-level designs presented.

5.4.1 Global Types and Electrical Units. As noted in C[illegible] III, an electrical units package and a global types package are used to define types that a[illegible] by all the packages within the system. The global types package is modified to include constant variables required to implement communications between the host and node processors on the iPSC/2. The electrical units package require the addition of a single record type that contains the voltage and LCF data as a single type. This is required to allow passing a single message for voltage and LCF on the iPSC/2. Also, constants that define the size of various types used in the PDESS are defined to save the overhead of function calls when determining the sizes of messages to be passed between processors.

5.4.2 Circuit Breaker, Bus and TRU Object Managers. The following methods are required for each object manager in the parallel design:

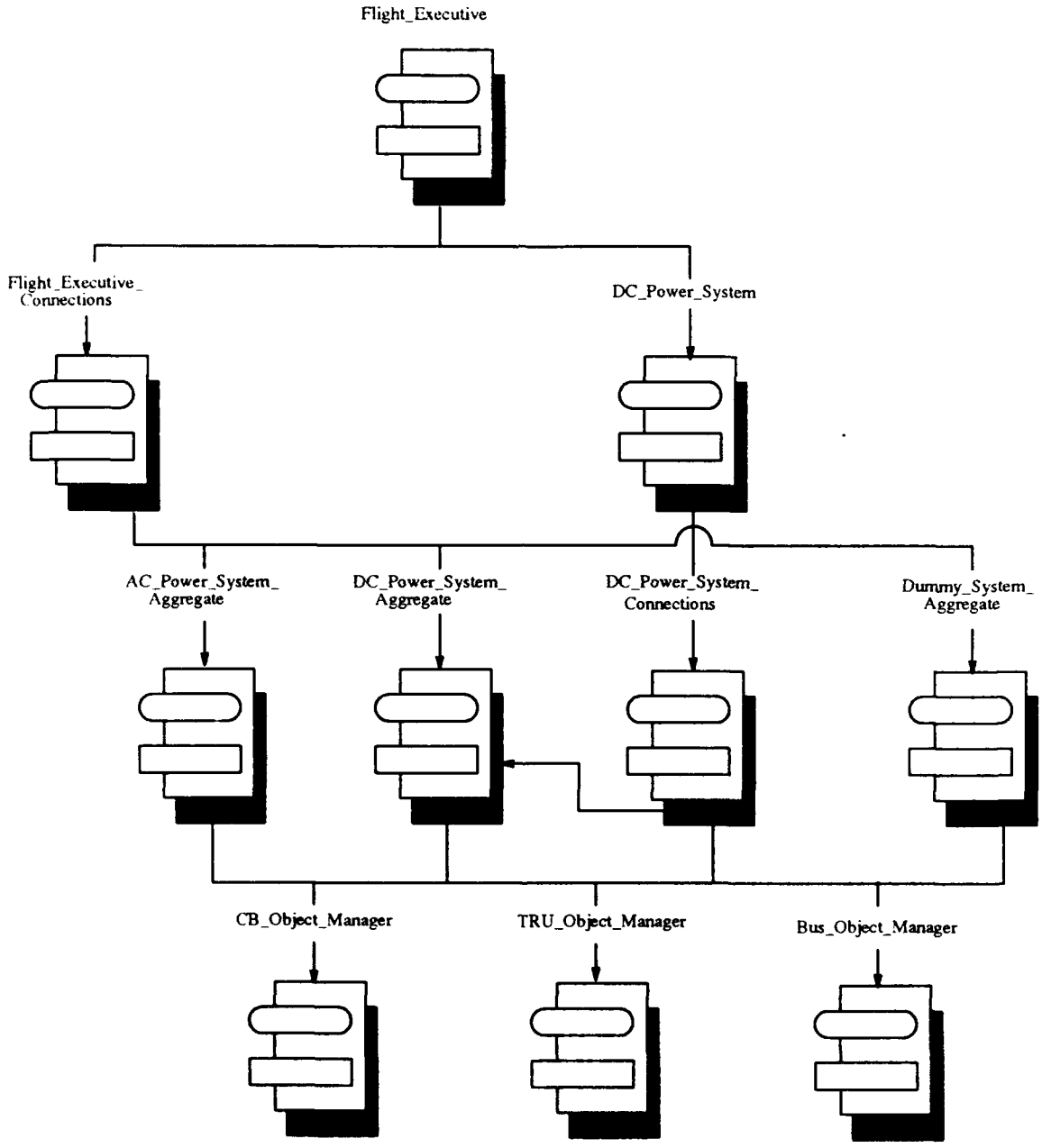


Figure 15. New Executive-Level Software Architecture

- Bus Object Manager
  - New_Bus
  - Delete_Bus
  - Give_Voltage_Lcf_To
  - Give_Current_To
  - Give_Power_Info_To
  - Get_Power_Info_From
  - Get_Number_Of_Points_From
- Circuit Breaker Object Manager
  - New_Cb
  - Delete_Cb
  - Give_Voltage_Lcf_To
  - Give_Current_To
  - Give_Power_Info_To
  - Get_Power_Info_From
  - Give_Position_To
  - Get_Position_From
- TRU Object Manager
  - New_TRU
  - Delete_TRU
  - Give_Voltage_Lcf_To
  - Give_Current_To
  - Give_Power_Info_To
  - Get_Power_Info_From

All the methods shown above, except the Delete_{object} method, were provided in the original sequential simulation. The Delete_{object} method is required to delete objects that are instantiated when the system aggregate packages are elaborated. The elaboration of objects by the system

aggregates will be described in more detail in the section on system aggregates. An exception for handling NULL object pointers is also required for the object managers, and is exported by each object manager.

5.4.3 DC Power System, System Aggregate, and Connections. The DC Power System exports the single procedure, `Update_DC_Power_System`. This procedure gates all the local and non-local intra-system connections in the DC Power System by using the system-level connection gating procedure. As was mentioned in Chapter III, the system connections are *not* implemented as separate packages. The system connections and their associated methods are contained within the system packages, and are depicted in the system architecture diagram of Figure 9 as separate packages for notational simplicity. Figure 15 follows the same method; and in making the design changes for the parallel version of the system connections packages, the system connections are still contained within the system packages. Thus, the DC Power System Connections are contained in the DC Power System package body.

The 110 system-level connections (connections 7 through 116) in the DC Power System (DCPS) are defined in the DCPS body. A DCPS connection type is defined as having a source, destination and a kind-of-data transferred. The kind-of-data is either voltage and LCF (VLCF) or current (load) data. The DCPS connections type is still defined as an array, as was done in the original DCPS package design, but the array is no longer a constant array. This is done to allow the node location variables for the connection points to be set at initialization time.

Another change from the sequential design is that a single procedure `Gate` is provided to gate all the system-level connections. This is different from the design of the sequential simulation where separate procedures are provided to gate VLCF, tie bus VLCF, current, and tie bus current connections. The parallel design provides a single procedure that gates all local and non-local VLCF and current connections using the gating algorithm defined previously in Section 5.3. The single `Gate` procedure is used instead of the original separate gating procedures because all connections

are logically equivalent in the new design, and a single procedure to gate all the connections is more appropriate for this design.

The system aggregate packages define the type `element_point` that is the source and destination type of a connection. Element points can be circuit breakers, TRUs, or buses, and each element point has a node location. These node locations are initialized by the `Initialize_Node_Locations` method provided in the body of each system package. The DCPS body executes its `Initialize_Node_Locations` method when the package is elaborated at execution time, as do the AC Power System and the Dummy System.[2] The `Initialize_Node_Locations` queries the DC Power System Aggregate to determine the location of the element point sources and destinations.

Another method defined in the body of the DCPS is a method that creates a local connection list array. This local connection list array is a subset of the connections enumerated in the DC Power System Connections array. It is simply a list of the connection numbers for all the connections that have a source or destination object located on a given node. Each node generates its own connection list at the time the DCPS package is elaborated during execution using the `Create_Connection_List` method. The `Create_Connection_List` procedure must execute after the `Initialize_Node_Locations` method is executed since it uses the node location information maintained in the DCPS connections array.

The specification for the DCPS Aggregate (DCPSA) package for the parallel design is identical with the specification used in the sequential version, except three changes. The first change is that the element point type now has a node location associated with it. The second change is that a location array for each type of object (Cb, TRU, and bus) is exported. These location arrays contain the node locations of all objects in the DCPS. The last change is that a method is exported for determining a given object's node location.

[2] The initialization procedure is called from within the execution section of the package when the package is loaded into memory for execution by another package that *with*s the first package. This provides a way for "automatic" initialization to be done by the *with*ed package without the using (or *with*-ing) package having to make a call to the initialization routine of the *with*ed package.

The sequential version of the DCPSA only has a specification. In the parallel design a DCPSA body is required for initializing the object node location information. The body of the package receives information from the host program with the location information for all objects in the simulation. The run-time initialization of object location information is done to allow easy reconfigurability of the simulation, and the initialization of all the system aggregates is done at the time the aggregate packages are elaborated.

5.4.4 AC Power System Aggregate, and Dummy System Aggregate. There are no AC Power System or AC Power System Connections packages in the sequential simulation or the parallel simulation; however, there is an AC Power System Aggregate (ACPSA) that names and instantiates the objects that are part of the ACPSA. The ACPSA specification is the same as the sequential version except that node location information and a method for obtaining the node location information are provided. This is similar to the design changes made in the DCPSA.

The Dummy System consists only of the Dummy System Aggregate (DSA) similar to the ACPS. The DSA package for the parallel design has the same changes that the other system aggregates have, i.e., node location information has been added the DSA element points and a method is provided to determine the location of any object that is part of the DSA.

5.4.5 Flight Executive. The flight executive package has the same design in the parallel and sequential versions. It exports the procedure `Update_Flight_Executive` that updates all the flight systems. The DESS only has one system that is fully simulated – the DC Power System. The AC Power System and the Dummy System are "stubbed-out." The AC Power System is defined by the AC Power System aggregate and the six AC buses that are the stubs for the AC Power System. The Dummy System is stubbed as ten circuit breaker objects defined in the Dummy System Aggregate. The objects for both these systems are initialized by the node simulation main program driver at the time the program starts execution. The sequential and parallel simulations contain the same objects and are stubbed the same.

5.4.6 Flight Executive Connections. The Flight Executive Connections (FECs) are a separate package from the flight executive. The FECs package has a design similar to the DC Power System Connections. The FECs are defined as an array of connection elements where each element is a system point type. Each system point is defined by an element point type from one of the three system aggregates. The node location information is maintained with the element points, and when gating a connection the Flight Executive Connection's `Gate` procedure queries the appropriate system aggregate to get the node location for that system's object.

The FEC package exports two procedures for operating on the executive-level connections defined in the FEC package. `Process_Cb_Linkages` is a method provided for processing external inputs for opening and closing circuit breakers. This method is also provided in the sequential version of the simulation. `Process_External_Connections_To_DC_Power` is the method called by the Flight Executive to gate all the executive-level connections for the DC Power System.

There are 16 flight executive connections defined in the Flight Executive Connections package specification. These connections are given different numberings than those given to the DC Power System connections so that all connections within the simulation can be uniquely identified by number. The FEC package body contains the `Gate` procedure for the executive-level connections. The connection gating algorithm is identical with the one used in the DC Power System Connections package, except that the connections gated are executive-level connections that have source and destination objects from three different systems.

The FEC body also contains a routine to initialize the location information of the system element points in the executive connection package. The node locations are initialized at elaboration time when the simulation starts running.

5.4.7 Node Simulation Program Driver. Each node that the simulation is running on will require a "main" procedure that calls other packages within the simulation and runs the simulation. To make the simulation easily scalable only a single main procedure is defined. This procedure

can run on any number of nodes, and uses the other packages defined above. The node main procedure initializes the objects defined in the AC Power System Aggregate and the Dummy System Aggregate, as is done in the sequential program. The procedure executes the number of iterations of the simulation provided by the user from the host main program.

5.4.8 Host Simulation Program Driver. A main program resides on the host system of the iPSC/2, and this program loads the main procedures for the iPSC/2 node processors. The main program reads the object map data files that define the location of all the objects in the PDESS. The host program sends this object map initialization data to each node, and sends the iteration count to each node.

5.5 Design Summary - Low-Level

The previous sub-sections described the low-level design of each package contained in the DC Electrical System Simulation. The differences between the parallel low-level design and the sequential low-level design were pointed out in the descriptions.

One thing that is apparent from the low-level design description is that most of the original sequential design is reused in the parallel design. The major changes that are required for the parallel version are adding location information to the system aggregates, and implementing a gating procedure for system-level and executive-level connections that can handle gating connections between objects on separate processors. The management of objects on nodes is simplified by there being only a single copy of an object on any processor. By using object maps provided by the user in a data file, and having each system aggregate initialize (or instantiate) its objects on each processor at run-time, the simulation can be easily scaled to various numbers of processors. This scalability and easy reconfigurability will aid in testing the simulation performance for various configurations of processors and object mappings. The next section addresses the performance issues of the simulation based on the low-level design.

5.6 *Low-Level Performance Analysis*

The design of the parallel version of the DESS (PDESS) is based on mapping one simulation object to each node in the parallel simulation, and each processor gating the local and non-local connections for the objects mapped to that processor. Each processor maintains a local topological list of connections that it gates each iteration of the simulation and each processor gates its non-local connections in the same relative, topological order to avoid dead-lock.

The goal of implementing the PDESS is to gain a speedup in the simulation execution time compared with the DESS. The way that objects are mapped to processors can have a significant effect on the potential speedup that can be achieved by the PDESS, and some heuristic for mapping objects to processors must take into account the factors of load imbalance, communications overhead and synchronization/processing-order dependencies mentioned in the concurrency analysis from Chapter IV.

There are eight processors on the AFIT iPSC/2 Hypercube, and there are 53 objects in the PDESS. If all possible mappings of objects to processors for one, two, four and eight processors are considered,[3] then the number of possible unique mappings of objects to processors is

$$1^{53} + 2^{53} + 4^{53} + 8^{53} \approx 8^{53} = 2^{159}. \tag{6}$$

This section presents an analysis of the low-level design. The goal of the analysis is to derive heuristics for selecting mappings that may achieve a speedup given that there are a large number of possible mappings of objects to processors for the PDESS (2^{159} possible mappings for the PDESS on AFIT's eight node iPSC/2 Hypercube). As mentioned before, there are three key factors that affect the potential speedup that can be achieved by any given mapping of objects to processors in the PDESS simulation.

[3] A hypercube architecture is constructed such that the processors are allocated in powers of 2. Thus, in using a hypercube, processors are allocated as 1, 2, 4, 8, ..., 2^n processors.

5.6.1 Load Imbalance. Connections are classified as local connections or non-local connections. Local connections have both their source and destination objects on the same processor, but non-local connections have only the source object or the destination object for a connection. Mapping objects to separate processors requires the transfer of state information between objects on separate processors when non-local connections are gated.

Gating a local connection (LC) involves reading state information from the side of the source object and applying that information to the side of the destination object. If the time that it takes to read state information from the side of an object is equal to T_{get} and the time that it takes to apply state information to the side of an object is T_{put}, then the time to gate a local connection is given by the equation:

$$T_{gate_{LC}} = T_{get} + T_{put}. \tag{7}$$

Gating a non-local connection on the processor that has the source for a non-local connection involves reading the state information from the side of the source object and sending the state information to the destination object. The time required to gate the non-local source connection (NLSC) on the source object processor is the sum of T_{get} and the time required to send the data, T_{send}. On the iPSC/2 Hypercube, T_{send} is simply the time it takes to post a non-blocking **isend** or a blocking **csend** message.[4] The time to gate a NLSC is given by the equation:

$$T_{gate_{NLSC}} = T_{get} + T_{send}. \tag{8}$$

Gating a non-local connection on a processor that has the destination object for the non-local connection requires reading the state information sent by the source object processor and

[4] An **isend** is an asynchronous, non-blocking send and a **csend** is a blocking send on the iPSC/2 Hypercube. For small messages sizes the **isend** or **csend** behave similarly and their execution times are equivalent.

applying the information to the side of the destination object using a `put_{attribute}` operation. Processing the non-local destination connection (NLDC) may involve time spent blocking for the incoming message from the source object processor because the blocking, topological-list connection processing method is used.[5] The time required to process a NLDC includes the time spent blocking for the incoming message, $T_{blocking}$, plus the time to read the received message once it has arrived, T_{recv}, plus the time required to write the received data to the destination object, T_{put}. The time to gate a NLDC is defined by the equation:

$$T_{gate_{NLDC}} = T_{blocking} + T_{recv} + T_{put}. \tag{9}$$

The total time spent by processor p gating connections for a single iteration of the simulation, T_{gate_p}, for L local connections, S non-local source connections and D non-local destination connections is defined by the equation:

$$T_{gate_p} = L * T_{gate_{LC}} + S * T_{gate_{NLSC}} + D * T_{gate_{NLDC}} + \text{Time spent blocking} \tag{10}$$

$$= L(T_{get} + T_{put}) + S(T_{get} + T_{send}) + D(T_{recv} + T_{put}) + \sum_{i=1}^{D} T_{blocking_i} \tag{11}$$

The execution time for the parallel simulation, T_e is equal to the maximum T_{gate_p} for the p processors used in the simulation, and is defined equationally as:

$$T_e = Max(T_{gate_1}, T_{gate_2}, \ldots, T_{gate_p}). \tag{12}$$

Load balancing for the PDESS is achieved by having equal values of T_{gate_p} for each of the processors used in the simulation. T_{gate_p} for each processor can be made equal by mapping objects to processors

[5] See Chapter IV, Section 4.3 for a description of the topological-list method.

such that the values of L, S, and D are equal for each processor. The mapping of objects should also be done to minimize the value of T_{gate_p}; thus, minimizing the value of T_e.

5.6.2 Communications Overhead. The PDESS incurs an additional workload that the sequential simulation does not incur due to the overhead of communications between processors. This communications overhead is due to the additional control logic in the parallel program necessary for executing send and receive messages between processors, and the transmission delay time involved in sending a message from one processor to another processor over the inter-processor communications link(s). The overhead introduced by these two factors increases the total execution time of the parallel processors, and thus reduces the potential speedup of the parallel simulation.

Communications overhead is a function of T_s, T_{recv} and $T_{blocking}$. T_{send} and T_{recv} are essentially constant values for the iPSC/2 for a given message size, as shown by experiments documented in [28]. $T_{blocking}$ is a function of the number of NLDCs on a processor and when messages are sent and received by NLSCs and NLDCs. If no non-local connections are present on a processor, then $T_{blocking}$ will be zero. $T_{blocking}$ can be equal to zero if all the objects in the PDESS simulation are mapped to single processor since all the objects within the PDESS are connected to each other through various series of connections, or $T_{blocking}$ can be equal to zero if the data for each non-local connection is received prior to the non-local connection being gated.

$T_{blocking}$ can be minimized by two methods. One method is to select a mapping of objects to processors that has a minimum number of non-local connections. The second method is to overlap communications between connections with connection computations. In the best situation for minimizing communications overhead, all NLSCs would be gated at the start of an iteration of the simulation, then all local connections would be gated, and lastly all NLDCs would be gated. Using this method, the computation of local connections could be executing while the messages from the NLSCs are being transferred between processors. If all the messages needed for the NLDCs arrived before the local connections gating was completed, then when the NLDCs are gated all

the data for each NLDC would be available and no blocking would be necessary when gating the connections. For the PDESS, this ideal overlap of communications and computations is limited by the fact that some LCs and NLSCs cannot be gated until some NLDCs have been gated. Therefore, only a few of the NLSCs can be gated prior to having to process a NLDC — which may have to block for incoming data to be received.

The amount of overlap that can be achieved in communications and computations is a factor of the topologically-ordered list that is being used by each processor, and this topological list is determined by how objects are mapped to processors. By selecting a mapping of objects that requires the minimum number of non-local connections and that provides the maximum amount of overlap in communications and computations, $T_{blocking}$ can be minimized and the potential speedup will be greater.

5.6.3 Processing-Order Dependencies of Connections. Some operations for the PDESS must be done sequentially due to the processing order dependencies of the connections in the simulation. As shown in Chapter IV, Section 4.3, some connections must be gated before other connections can be gated. Given n connections that must be executed in order from 1 to n where each connection takes a single unit of time to gate, then gating one connection on each of n processors will require n units of time due to the processing-order dependencies of the connections. If the n connections can be processed in any order, then for n processors the gating of the connections could be done in one time unit. The time required to gate the n sequentially-dependent connections is equal to n units of time because the processor with connection $n + 1$ must wait until the processor with connection n has gated its connection. Thus, for a single iteration of gating n connections that must be processed in order, no speedup can be gained by using more than 1 processor. In fact, gating the connections by more than one processor could be slower than using only one processor due to communications overhead between processors.

The potential for an order n speedup is possible if more than one iteration of the gating of the n connections is done using n processors and if there is no feedback between connections. This speedup is possible due to a "pipeline-effect" that can be achieved by each processor gating its connection every unit of time after the first time that its $n-1$ predecessor connection is gated. For example, if there are 3 connections to be gated on 3 processors for i iterations, then at time T_1 connection 1 is gated. Connection 1 and 2 are gated at time T_2, and connections 1, 2 and 3 are gated at T_3. Iteration 1 is finished after connection 3 is gated at T_3, and iteration 2 is finished after T_4. Iteration i is finished after time T_{i+3}. For n connections gated by n processors for i iterations, the speedup that can be achieved using the pipeline-effect is

$$S = n\left(\frac{i}{i+n}\right) \tag{13}$$

$$\approx n \quad \text{if} \quad i \gg n. \tag{14}$$

Thus, for a large number of iterations the speedup achievable can be of order n if the pipeline-effect can be used in the gating of sequential connections.

The order n potential speedup using the pipeline-effect is not achievable in the PDESS simulation due to the synchronization required between iterations of the simulation. The synchronization between iterations is due to the nature of how messages are passed between objects within the simulation. For every connection that is gated to send VLCF data from one object to another, there is a complementary connection that sends current in the opposite direction. The $n+1$ iteration gating of a VLCF connection cannot be executed until the n iteration load/current connection is gated. For example, if a message is sent from processor P1 to processor P2 with the value of VLCF from an object on P1, then a message will be sent back from P2 to P1 with the value of the current from the object on P2 that received the VLCF from P1. At time T_1, P1 can gate its VLCF connection to send VLCF data to P2. If the transfer time for sending VLCF data from P1 to P2 is considered as part of T_1, then at time T_2, P2 can gate its current connection to send current

data to P1. The VLCF connection of P1 cannot be gated at T_2 because P1 must wait until P2 has gated its current connection before P1 can gate its VLCF connection for the second iteration. This connection gating order is a due to the way the state of each object in the simulation is calculated. If P1 gated its VLCF connection at T_2 (the same time that P2 was gating its current connection) then the objects on P1 and P2 would be in an inconsistent state at time T_3. At time T_2 only P2 can gate its current connection and P1 must wait until T_3 to gate it VLCF connection. Thus, the pipeline-effect cannot be used to gain any increase in speedup for processing of serially dependent connections during multiple iterations of the PDESS.

5.6.4 Mapping Heuristic. Load-balance, communications overhead, and serial execution of connections are all factors that affect the potential speedup of the PDESS. The mappings of objects to processors that have the highest potential for speedup are mappings that:

- minimize load-imbalance,
- reduce communications overhead by overlapping communications and computations, and
- keep serially-dependent connections on the same processor.

Mappings of objects to processors are selected for testing by choosing mappings that generate roughly equal numbers of local and non-local connections for each of the n processors used in each test case. Also, the mappings are selected such that the topological-list for each processor provides a good degree of overlap of communications and computations. This should reduce the value of $T_{blocking}$ for each processor and aid in gaining speedup. Lastly, mappings are chosen that do not distribute serially-dependent connections between processors. Figures 16 through 23 show the mappings that are tested to measure speedup in the next chapter.

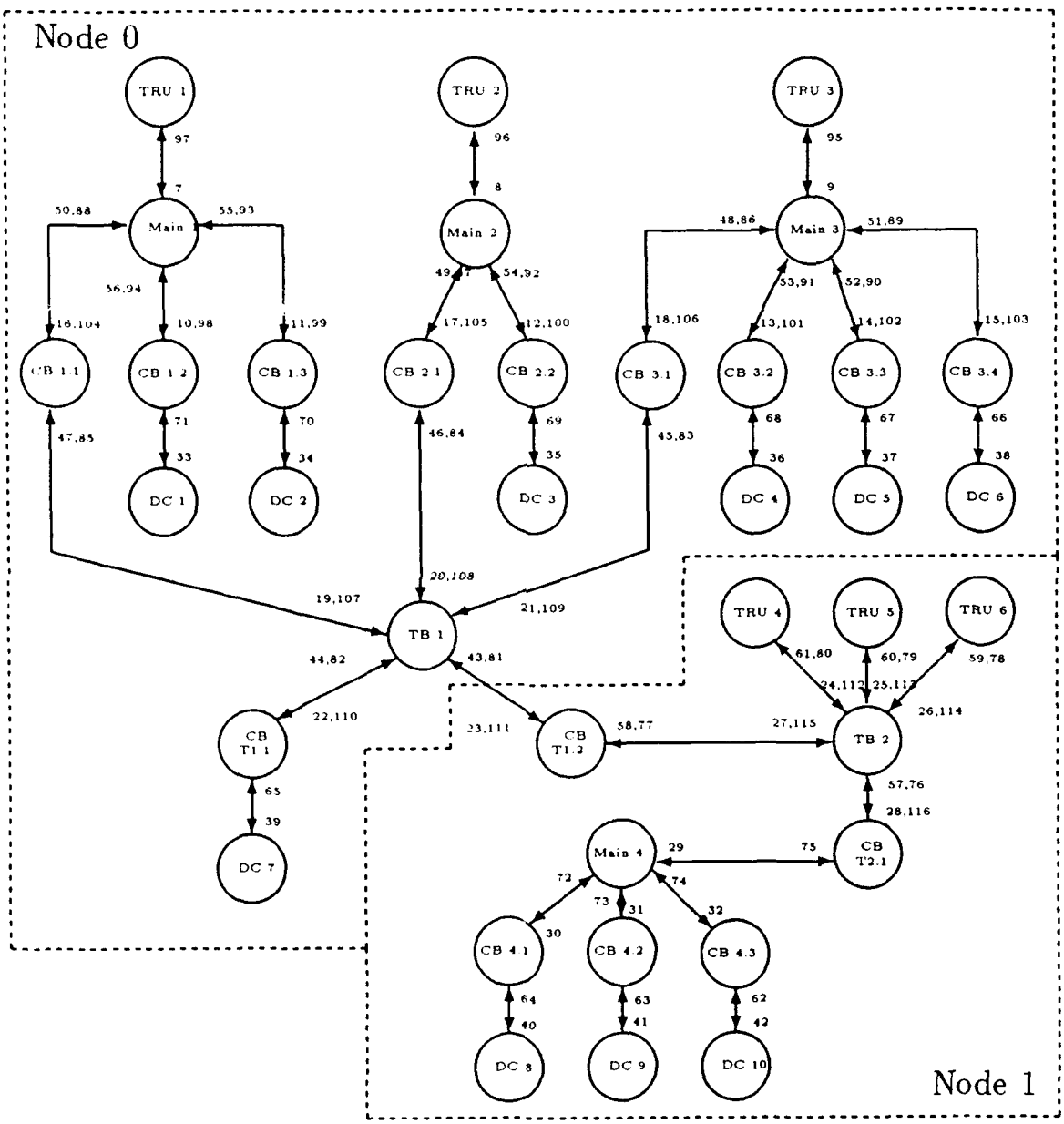


Figure 16. Mapping of the PDESS objects for configuration 2v1.

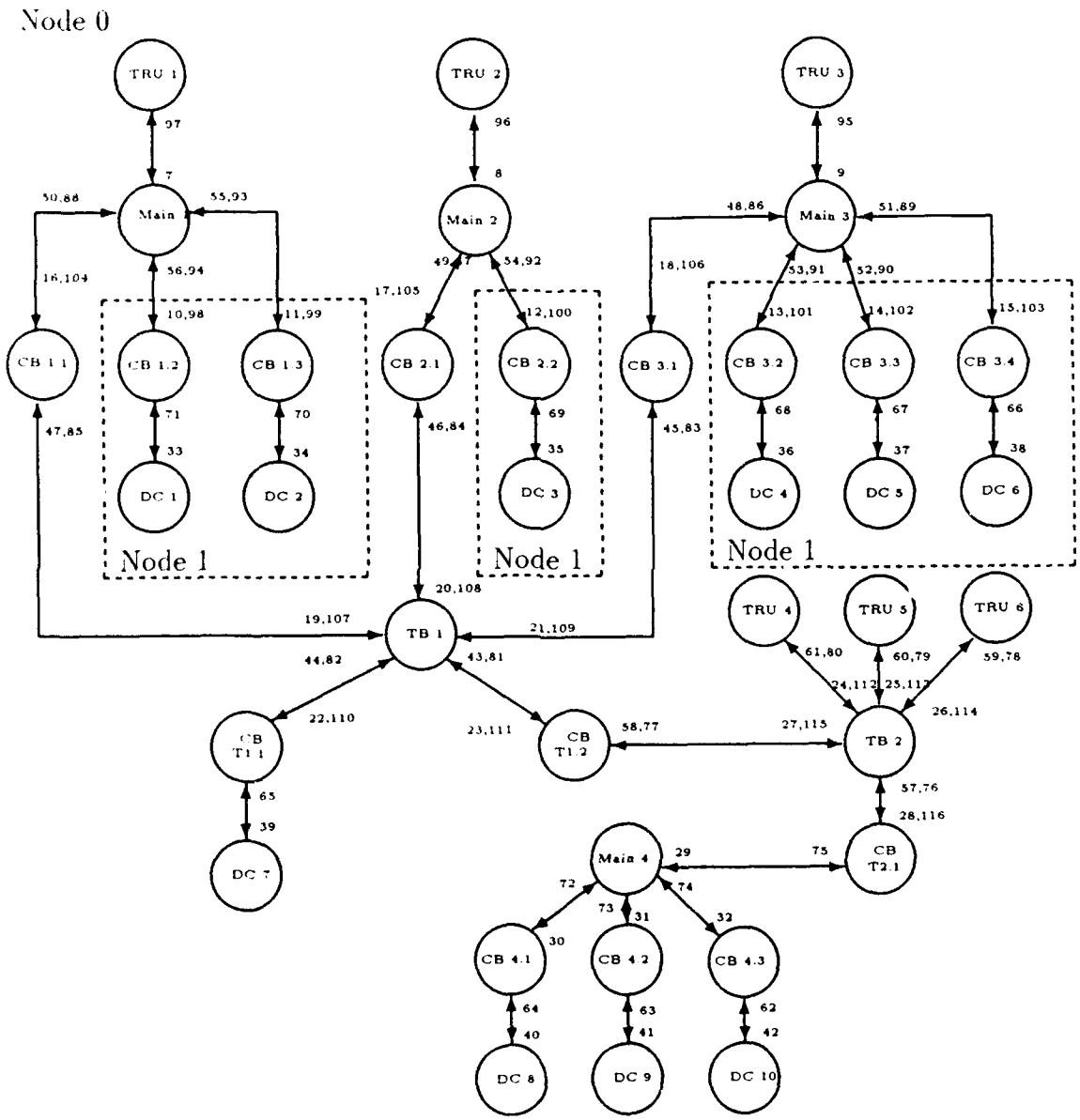


Figure 17. Mapping of the PDESS objects for configuration 2v2.

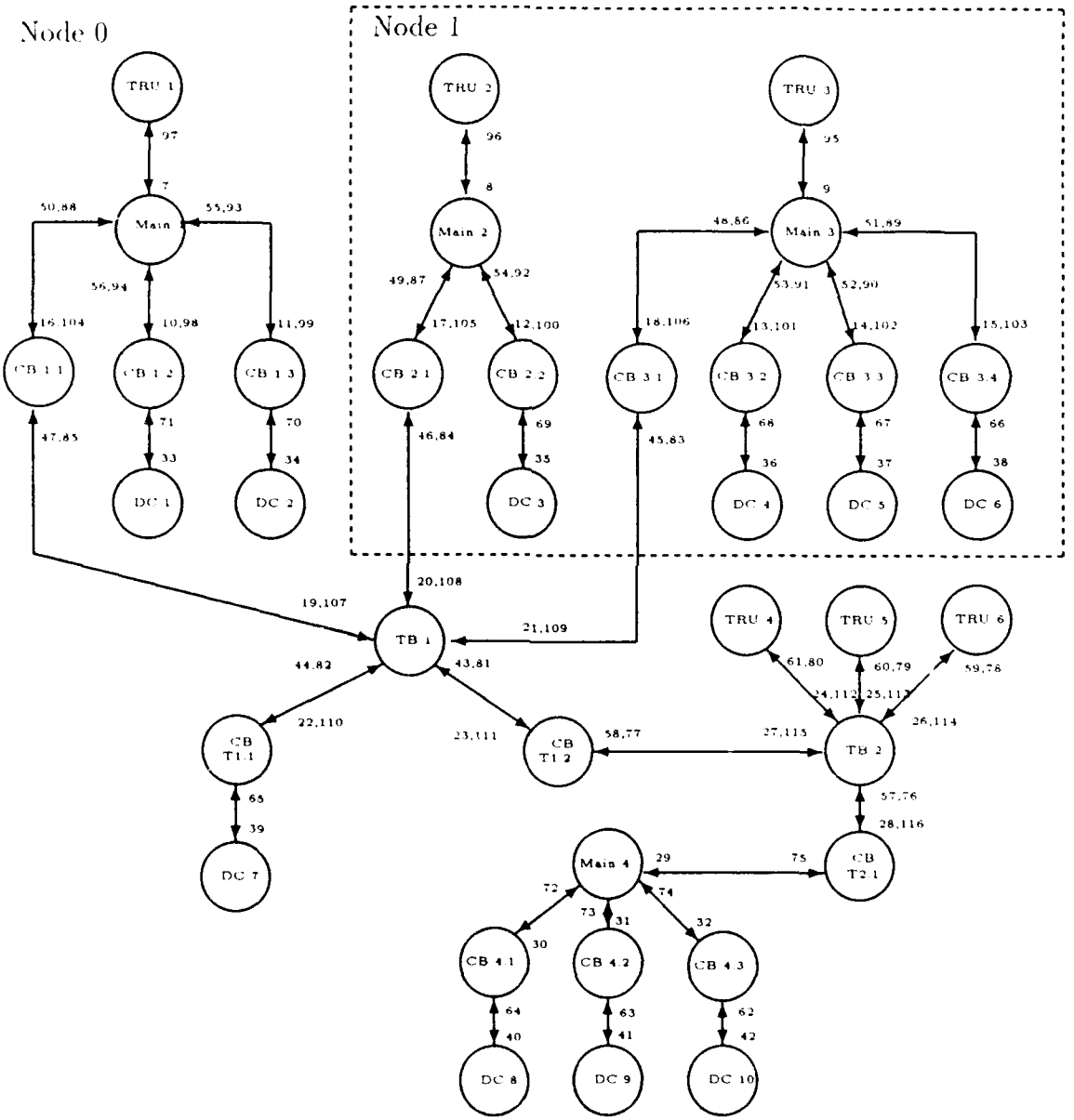


Figure 18. Mapping of the PDESS objects for configuration 2v3.

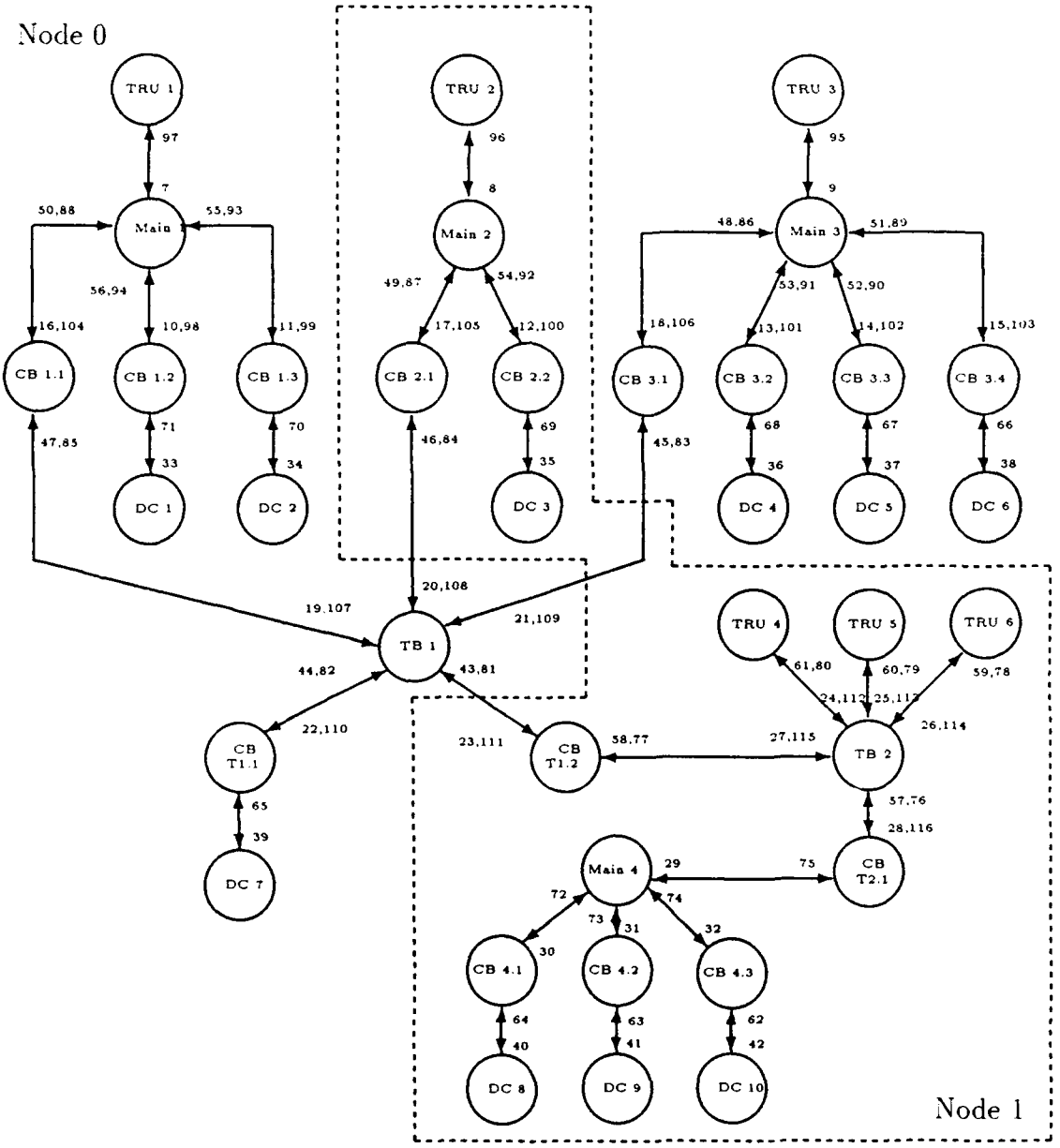


Figure 19. Mapping of the PDESS objects for configuration 2v4.

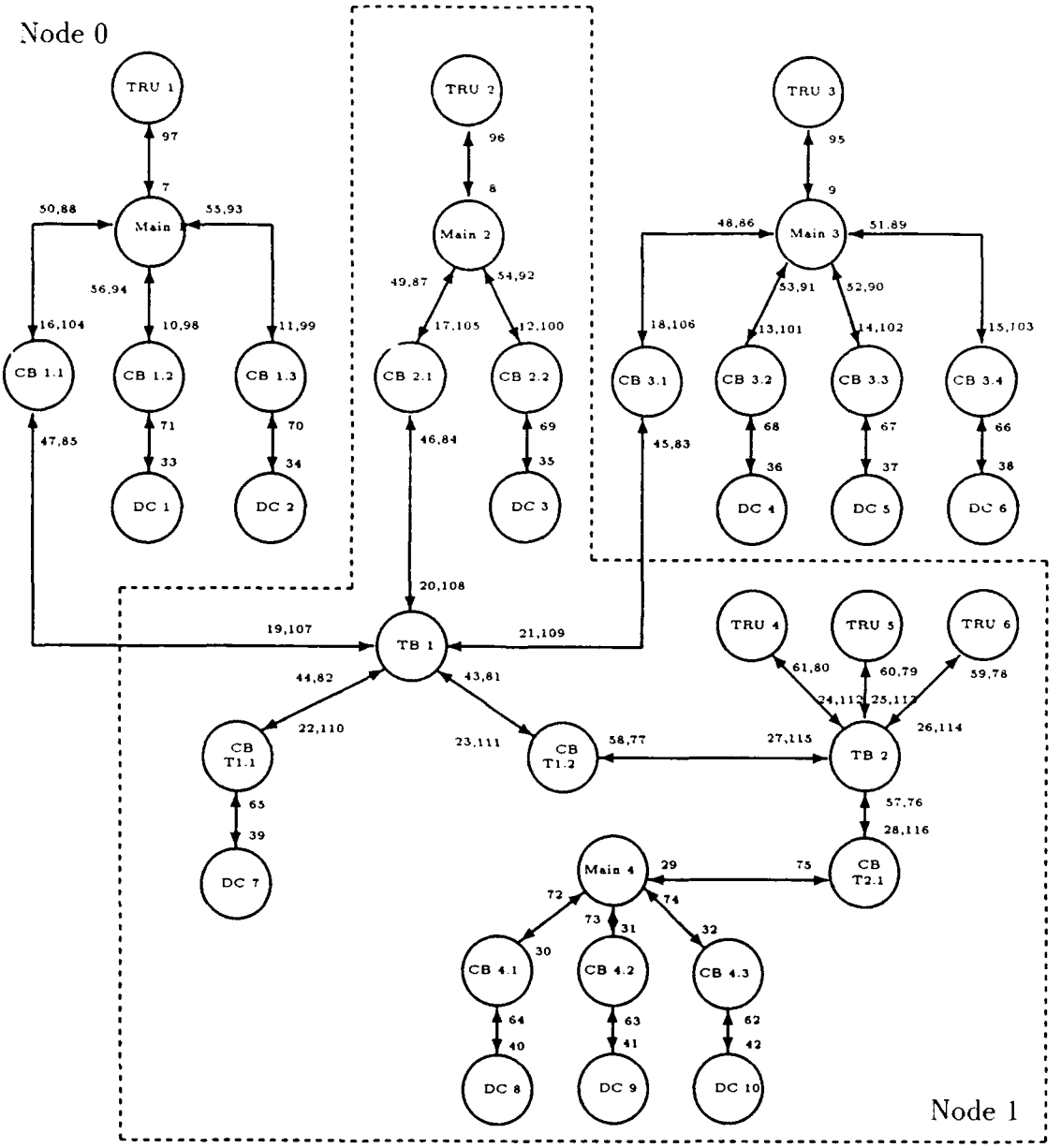


Figure 20. Mapping of the PDESS objects for configuration 2v5.

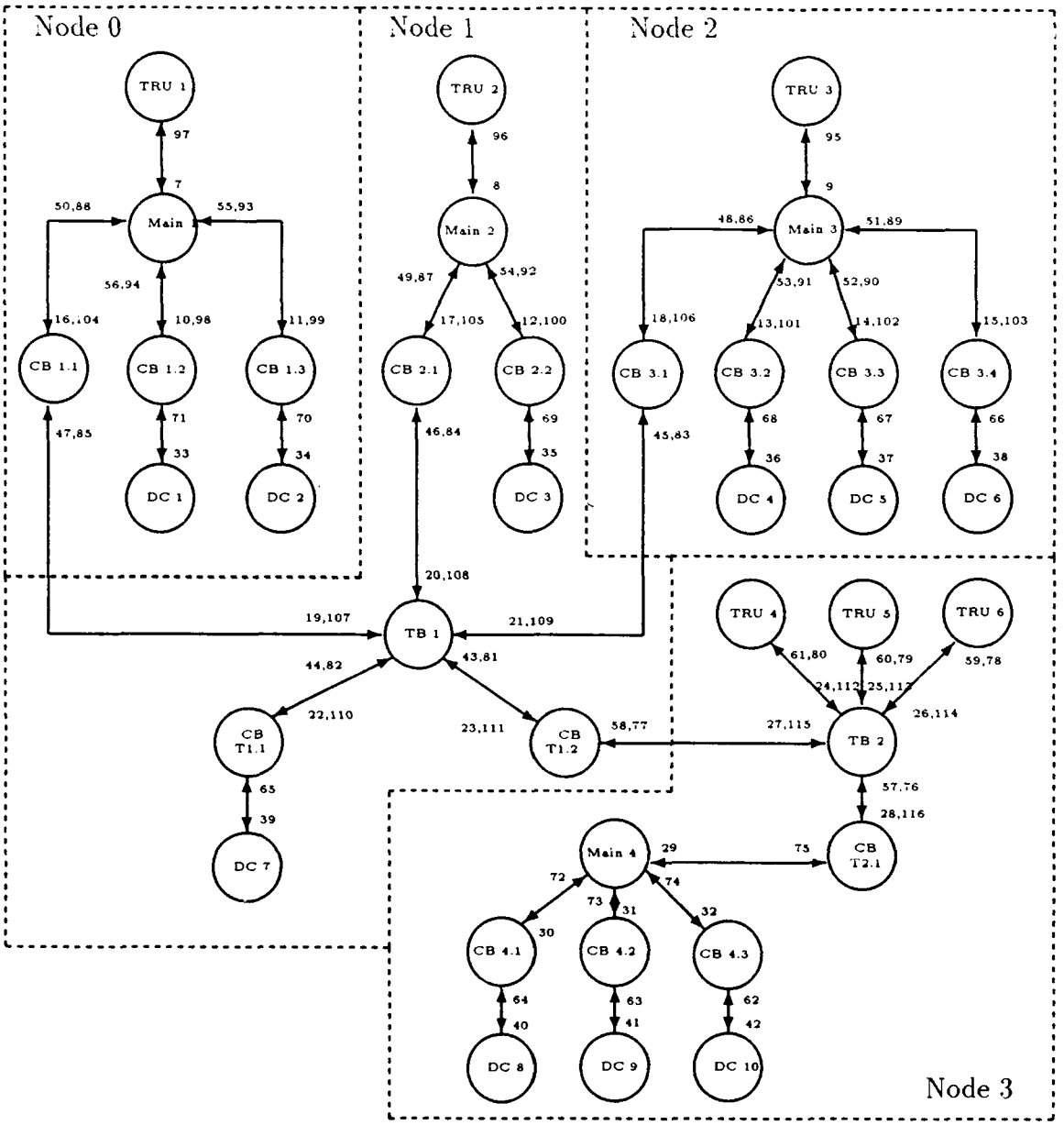


Figure 21. Mapping of the PDESS objects for configuration 4v1.

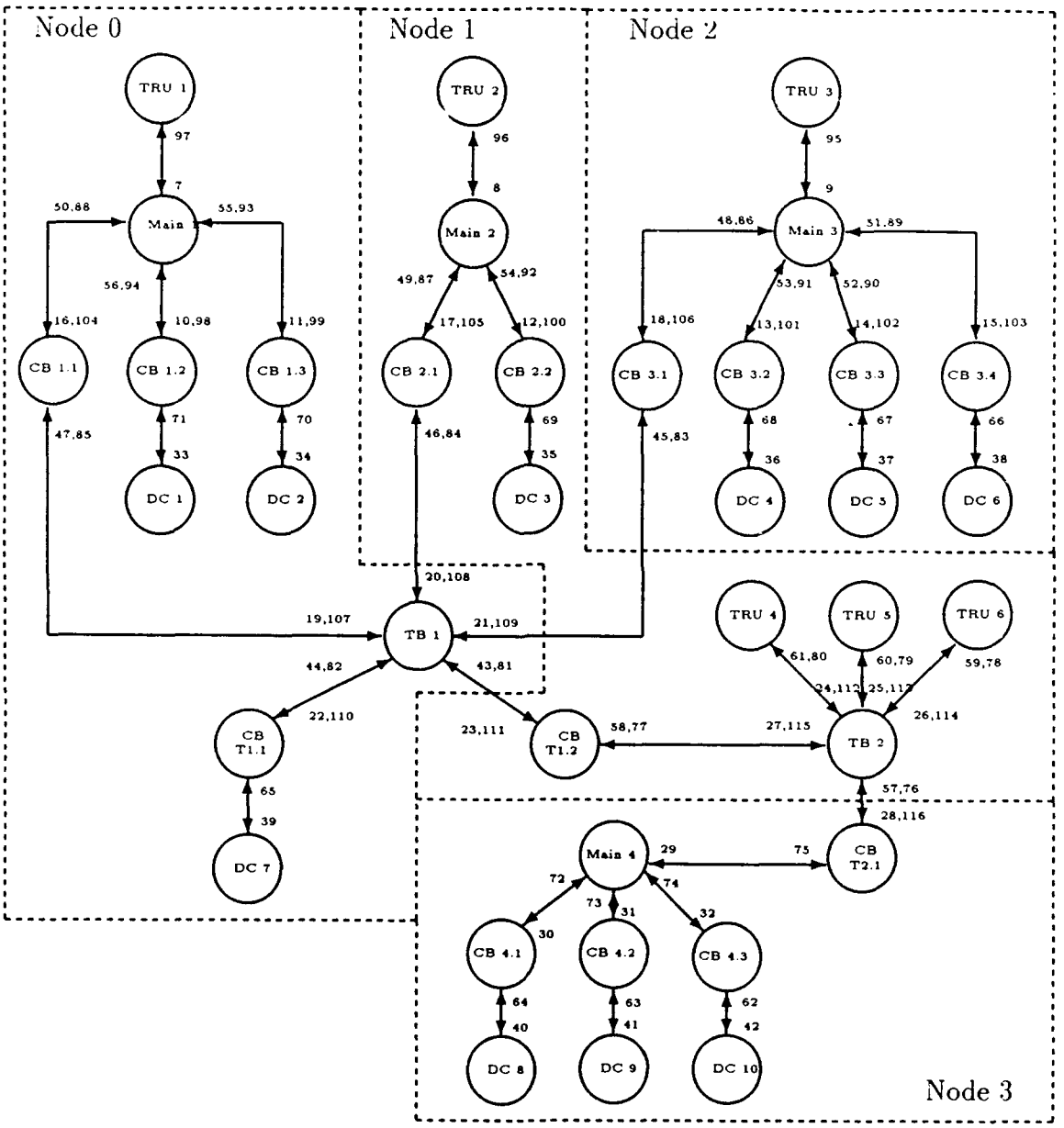


Figure 22. Mapping of the PDESS objects for configuration 4v2.

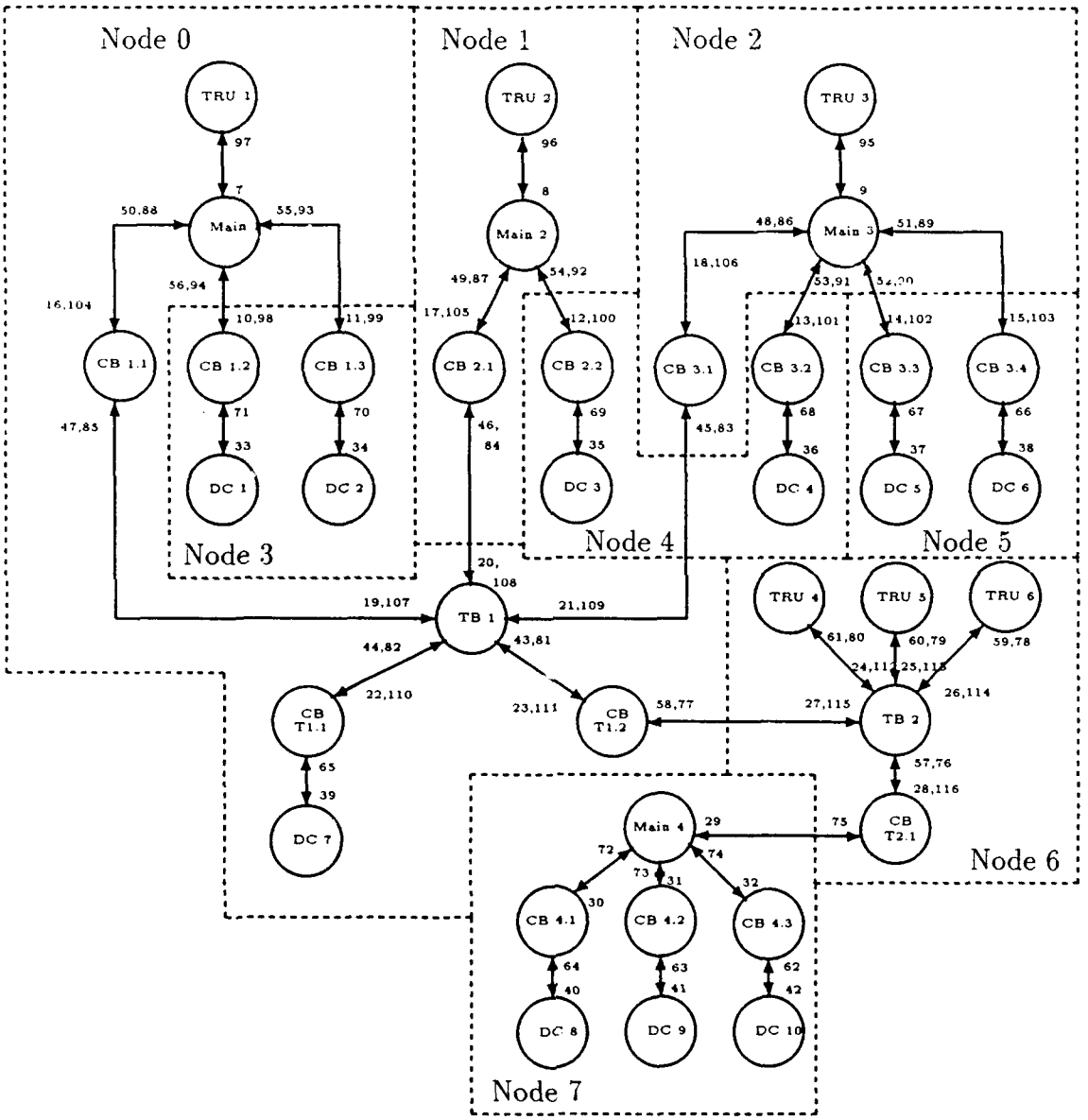


Figure 23. Mapping of the PDESS objects for configuration 8v1.

VI. Performance Results of the Parallel OOD Simulation

6.1 Introduction

The first section of this chapter describes how the parallel simulation was validated to make sure that it generates the same results as the sequential simulation. The next section describes the procedures used for testing, and in the last section the performance testing results for the Parallel DC Electrical System Simulation (PDESS) are presented.

6.2 Validation of Results

The PDESS was tested to determine whether it generated the same outputs as the sequential simulation. To validate the results, the sequential simulation was run for 1, 2, 5, 10, 100 and 1000 iterations. The state of all the objects in the simulation was printed to a file at the end of each of these simulation runs, and these files were sorted based on the first field in the files, which was the object name.

The PDESS was run for 1, 2, 5, 10, 100 and 1000 iterations on one, two, four, and eight processors. The state of the objects on each node was sent to a file at the end of each of these runs. The state information output from the PDESS was in the same format as the sequential version to aid in comparing the parallel and sequential outputs. The parallel outputs were sorted in the same manner as the sequential files, and then the same iteration output files for the sequential and parallel simulations were compared.

In the first series of tests, the outputs of the PDESS and the sequential simulation were different. This difference was traced back to an improper handling of tie buses connection gating. This error was corrected, and the parallel simulation was run again. The outputs were compared again, and the state of all the objects in the simulation was identical for all runs of the parallel and sequential simulations. Various mappings of objects to processors were tested, and the results of the

tests yielded the same states for all configurations. Thus, the parallel simulation was empirically validated and shown to generate the same results as the sequential version of the simulation.

6.3 Performance Tests

6.3.1 Speedup Calculations. The speedup of the PDESS is measured by comparing the best execution time for n iterations of the sequential simulation against the best execution time for n iterations of the parallel simulation. The formula for speedup, S, is calculated by the equation,

$$S_p = \frac{\text{Best time for } n \text{ iterations of sequential program}}{\text{Best time for } n \text{ iterations of parallel program}} \quad (15)$$

Initialization time is not included as part of the execution time for each of the simulations since the PDESS has a high initialization overhead due to its reconfigurability feature. The reconfigurability feature is built into the initialization of the simulation to allow rapid testing of different configurations of processors and object mappings. As described in Chapter V, the host program reads a configuration file that identifies where each simulation object is mapped, and this mapping information is sent to each node. This initialization overhead can be reduced by setting the node location variables in each system aggregate to a fixed location and removing the initialization routines from the host program. This will speed up initialization, but will make reconfiguration of the simulation a tedious task that requires recompiling the system aggregates for each change in the configuration. Therefore, the PDESS initialization is not compared with the sequential initialization time in calculating speedup since the PDESS initialization procedures are not optimized for speed of execution.

The execution time for the parallel simulation is determined by having each processor report the total execution time for the total number of iterations. The number of iterations that the simulation is to run is an input parameter provided by the host to the nodes. Each node runs the same number of iterations of the simulation. The total execution time for the parallel simulation is

the longest execution time of any of the processors, since all the processors start execution at the same time.

6.4 Timing Test Procedures

The execution times were measured using the node operating system clock function **mclock** — a clock function with 1 millisecond resolution [26]. The **mclock** function was called at the beginning of a series of iterations (after all initialization of the nodes was completed) to get time T_{start} on each node. After all the iterations were completed on each node the **mclock** function was called again to get a value for time T_{finish}. The execution time for each node was calculated as,

$$T_{e_p} = T_{finish} - T_{start} \qquad \text{where } p \text{ is the processor number.} \tag{16}$$

The execution time of the parallel simulation is equal to the maximum execution time of of all the processors. The execution times of the simulation have an error of $\pm$ 1 millisecond due to the resolution of the clock and the way the execution time is calculated based on two readings of the **mclock**.

All the timings reported were done with no other processes running on any of the unused nodes of the hypercube, e.g., for the two node test runs only two of the nodes were used for the testing, but the other 6 nodes were left running idle. This was done to eliminate increases in execution time that were seen in timing results that were run when other processes shared the hypercube. When the hypercube was dedicated to running only the test programs, the variability of timings was less than two percent between runs.

The amount of time spent in blocking during the execution of the **crecv** operations was measured for each node also. This was done to get an idea of the values of $T_{blocking}$ for each configuration tested. The amount of time spent blocking by each node was measured by using the **mclock** function before and after each **crecv** operation used when gating a connection. The

Table 5. AC Power System Object to Processor Mappings

Configuration	1v1	2v1	2v2	2v3	2v4	2v5	4v1	4v2	8v1
Total No. of Processors	1	2	2	2	2	2	4	4	8
AC Bus 1	0	0	0	0	0	0	0	0	0
AC Bus 2	0	0	0	1	1	1	1	1	1
AC Bus 3	0	0	0	1	0	0	2	2	2
AC Bus 4	0	1	0	0	1	1	3	1	6
AC Bus 5	0	1	0	0	1	1	3	1	6
AC Bus 6	0	1	0	0	1	1	3	1	6

Table 6. Dummy System Object to Processor Mappings

Configuration	1v1	2v1	2v2	2v3	2v4	2v5	4v1	4v2	8v1
Total No. of Processors	1	2	2	2	2	2	4	4	8
Dummy Cb 1	0	0	1	0	0	0	0	0	3
Dummy Cb 2	0	0	1	0	0	0	0	0	3
Dummy Cb 3	0	0	1	1	1	1	1	1	4
Dummy Cb 4	0	0	1	1	0	0	2	2	4
Dummy Cb 5	0	0	1	1	0	0	2	2	5
Dummy Cb 6	0	0	1	1	0	0	2	2	5
Dummy Cb 7	0	0	0	0	0	1	1	0	0
Dummy Cb 8	0	1	0	0	1	1	3	3	7
Dummy Cb 9	0	1	0	0	1	1	3	3	7
Dummy Cb 10	0	1	0	0	1	1	3	3	7

value reported for $T_{blocking}$ on each node was the cumulative time spent blocking for all the `crecv` operations. If the message that the `crecv` was called to read was available on the processor, then the time measured for executing the `crecv` function was less than the resolution of the `mclock` function, and the reported blocking time for that connection was zero. However, if the `crecv` function blocked for one millisecond or more then the blocking time for that connection was measured with an accuracy of $\pm$ 1 msec. Due to the resolution of the clocking function and the way that the blocking time was accumulated, the accuracy of the total time reported for $T_{blocking}$ for a node is equal to $\pm(i * D)$ msecs, where i is the number of iterations and D is the number of non-local destination connections. Thus, the values reported for $T_{blocking}$ are good estimates, but may have a wide variability due to the resolution of the clock, and the way that $T_{blocking}$ is calculated. The values should still give an indication of the magnitude of the time spent blocking by each processor.

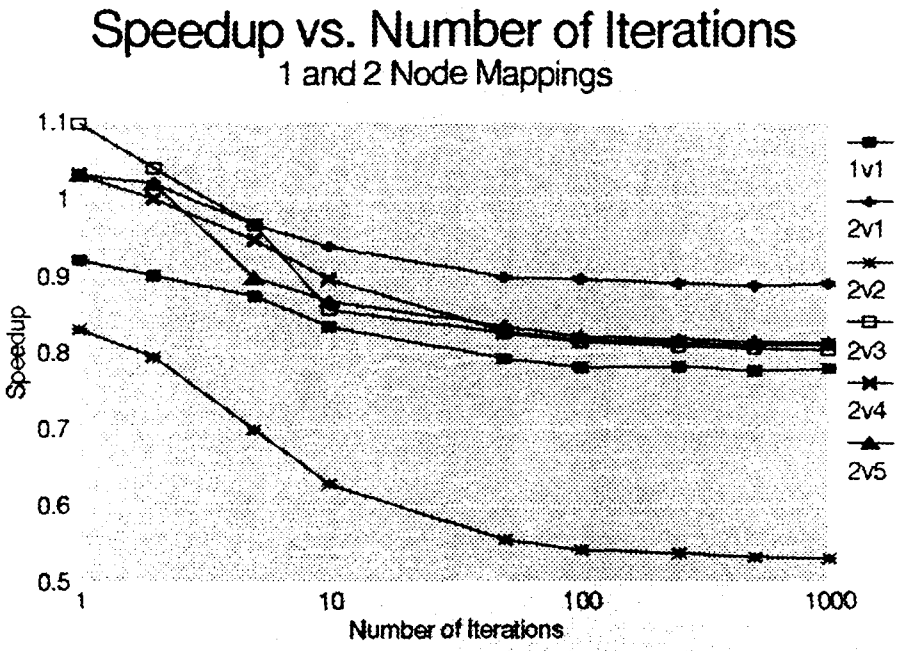
6.5 *Timing Test Results*

The first series of timing tests were conducted with 2, 4 and 8 processors. Each of the DC Power System object configurations shown in Figures 16 through 23 was tested, and the AC Power System and Dummy System objects were mapped as shown in Tables 5 and 6. The tables show that the AC Power System and the Dummy System objects are mapped to the same processor that their executive-level connection destination objects from the DC Power System are mapped. Thus, all executive-level connections are local connections for all tested configurations.[1] The execution time for each of the n processors was measured for 1, 2, 5, 10, 50, 100, 250, 500 and 1000 iterations for each of the mapping configurations.

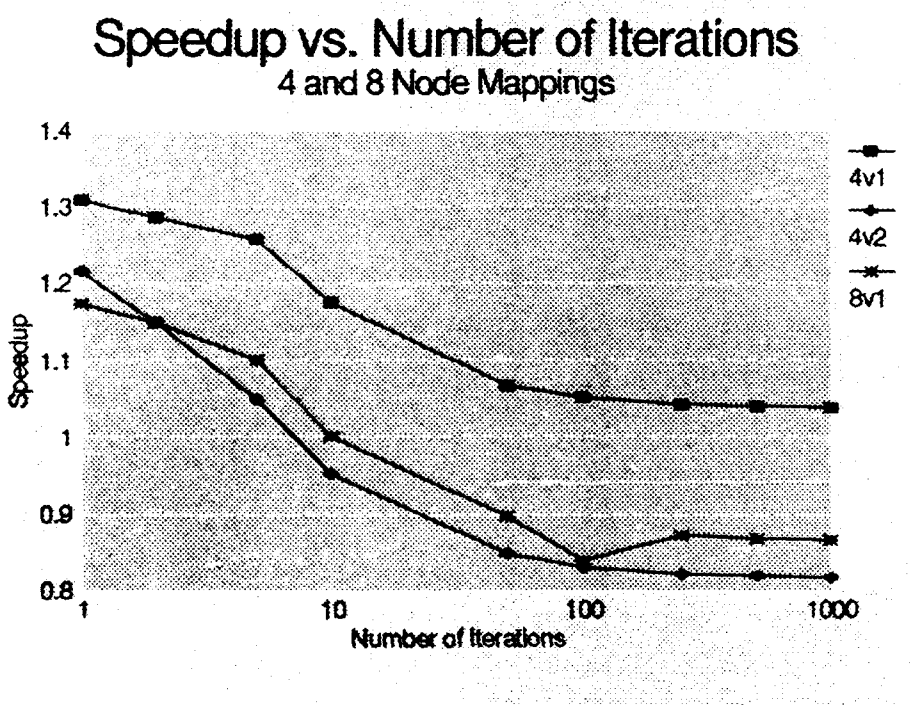
The speedups measured for the first series of timing tests are shown by the two graphs in Figure 24. Both graphs indicate that the speedup for all the configurations decreases with the number of iterations run. There is a significant decrease in the speedup of all configurations of the simulation in going from one to 100 iterations. The speedup stays approximately the same in going 100 to 1000 iterations. All the 2 node configurations show a slowdown for five or more iterations. Only the four node configuration 4v1 shows a speedup after 10 iterations, but the speedup for 4v1 goes from 1.3 to less than 1.15. This speedup is a very small increase in performance given that four processors are being used, and the slowdown for all the other configurations tested is obviously not an encouraging result.

Additional testing is necessary to determine the reason that the speedup decreases with the number of iterations of the simulation, and to determine what factors cause the slowdown for all the configurations except 4v1 — which only has a small speedup. In order to measure the maximum speedup that can be achieved by any of the given object mapping configurations, a series of timing

[1] The executive-level connections package can support non-local connections gating in the same manner that the system-level connections package does, but inefficient mappings of objects to processors would be generated if the objects from the AC Power System and the Dummy System are mapped to processors such that non-local executive connections are required. This condition is due to the AC Power System and the Dummy System being "stubbed-out." A simulation with a complete AC Power System and a Dummy System may introduce non-local executive-level connections.



(a)



(b)

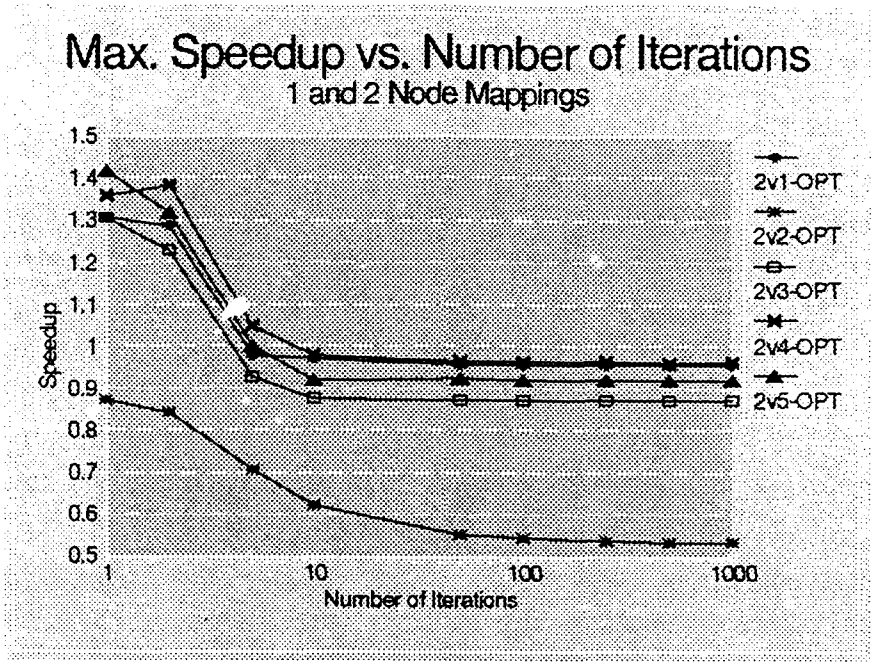
Figure 24. Measured Speedups of (a) 1 and 2 Node Object Mappings, (b) 4 and 8 Node Object Mappings.

tests is run with a modified version of the PDESS. The DC Power System package body of the modified PDESS is changed so that all the NLSCs are gated first, then the local connections are gated, and lastly the NLDCs are gated. This method of gating the connections on each processor results in the maximum level of overlap of communications and computations, and indicates the highest level of speedup that is possible with a given mapping of objects to processors. The actual states of the objects generated using this modified version of the PDESS is not the same as the sequential simulation since the connection gating dependencies are not handled correctly when the connections are gated in this modified manner. However, this method of gating the connections gives an upper bound to the speedup that can be achieved by a particular mapping of objects to processors using the PDESS.

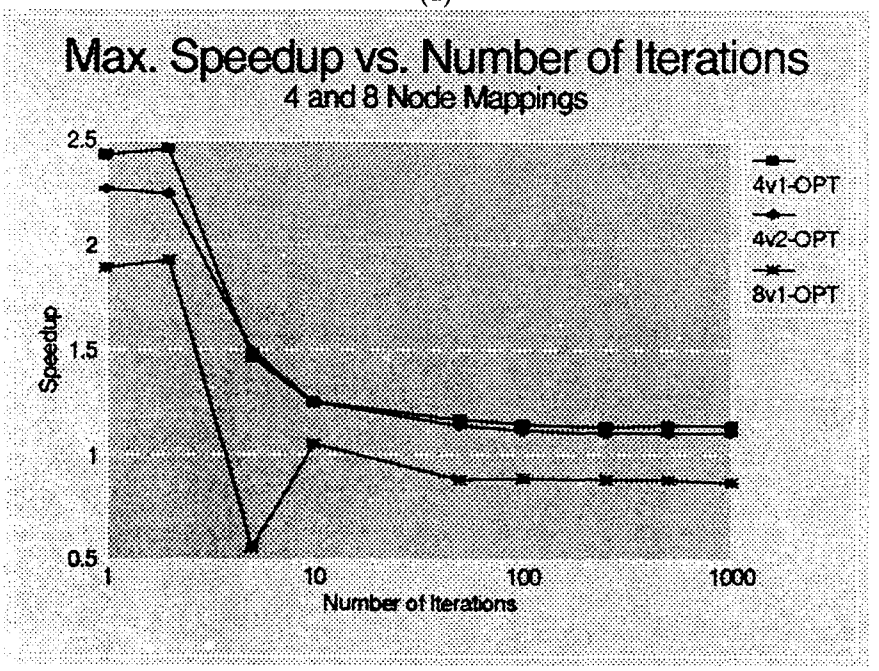
Figure 25 shows the speedup obtained by using the modified PDESS. Figure 25(a) shows the speedup for all the two node configurations using the modified PDESS, and Figure 25(b) shows the speedup for the four node and the eight node configurations. For each of the configurations (except configuration 2v2), the speedups for one and two iterations are greater than the unmodified PDESS speedups; but as the iterations increase toward 1000 iterations the speedups of the modified and the unmodified PDESS are almost identical. For the 2v2 configuration, the unmodified PDESS speedup is equal to the modified PDESS speedup — which is actually a slowdown. Both 2v2 configurations show the poorest performance of all the configurations.

The modified PDESS simulation speedups decrease with an increase in the number of iterations similar to the decrease seen in the unmodified version of the PDESS. For a small number of iterations, a much higher speedup is measured than when the number of iterations goes beyond 5 or 10 iterations. This decrease in the speedup for both the unmodified and the modified PDESS is an interesting phenomenon, and the next section addresses the cause of this phenomenon.

6.5.1 Variable T_{get} Times. Some measurements were done to determine the relative time involved in gating a connection. It was found that a large segment of the time involved in processing



(a)



(b)

Figure 25. Maximum Speedups of (a) 2 Node, and (b) 4 and 8 Node Object Mappings.

a connection is reading the state of a connection. This is a large part of the time of gating a local connection (LC) and a NLSC because the new state of an object is calculated when the object's state is read by using the `Get_Power_Info_From` method. The `Get_Power_Info_From` method is the first operation done when gating a LC or an NLSC. As was noted in Chapter III, the state of an object can be computed when an object receives inputs on a side or when the state of the object is read. This is a design choice; the paradigm does not define when the state should be calculated. The sequential simulation calculates the new state when the `Get_Power_Info_From` method is applied to an object. Since the PDESS uses the object-manager design from the original sequential simulation, new states of objects in the PDESS are also calculated when the state of an object is read using the `Get_Power_Info_From` method.

Calculating the new values of voltage-LCF, and current that are required when the `Get_Power_Info_From` method is applied to a bus object involves time consuming floating point operations.[2] The number of floating point calculations done for a bus object has an order of magnitude proportional to the number of bus connections, e.g., `Get_Power_Info_From` has $O(n)$ where n is the number of bus connections. The time that it takes to execute the `Get_Power_Info_From` method on a bus is also affected by whether the values applied to the side of the bus are different from the current values of voltage and LCF, or current. If the values applied to the side of a bus by a `Give_{attribute}` method are the same as the previously applied values, then when the `Get_Power_Info_From` method is executed, the floating point calculations are not executed since the side inputs have not changed. This makes the execution time of the `Get_Power_Info_From` method variable; thus, T_{get} for buses varies. The execution time of gating a connection can vary significantly based on whether the state of the inputs to the side of a bus are changed or not. From measurements done on bus objects, the amount of time that it takes to execute the `Get_Power_Info_From` method on a bus can vary

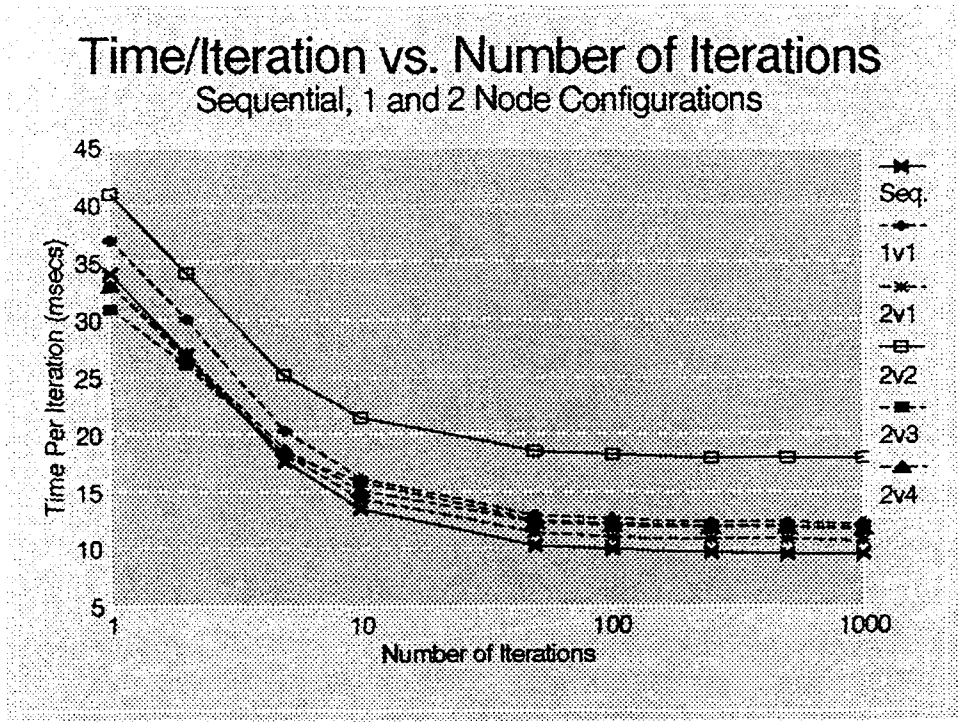
[2]Floating point operations typically take significantly longer than integer operations for most microprocessor architectures. Even with the 80387 math coprocessor that the iPSC/2 uses for floating point operations for Ada, floating point operations are very time consuming. See [28] for some performance measurements for the iPSC/2 floating point operations.

from 5 milliseconds to less than 1 millisecond for a five-connection bus — the size of the largest buses in the PDESS. This variable T_{get} time for buses and TRUs can cause a significant change in the computational workload from one iteration to the next in the PDESS and the sequential simulation.

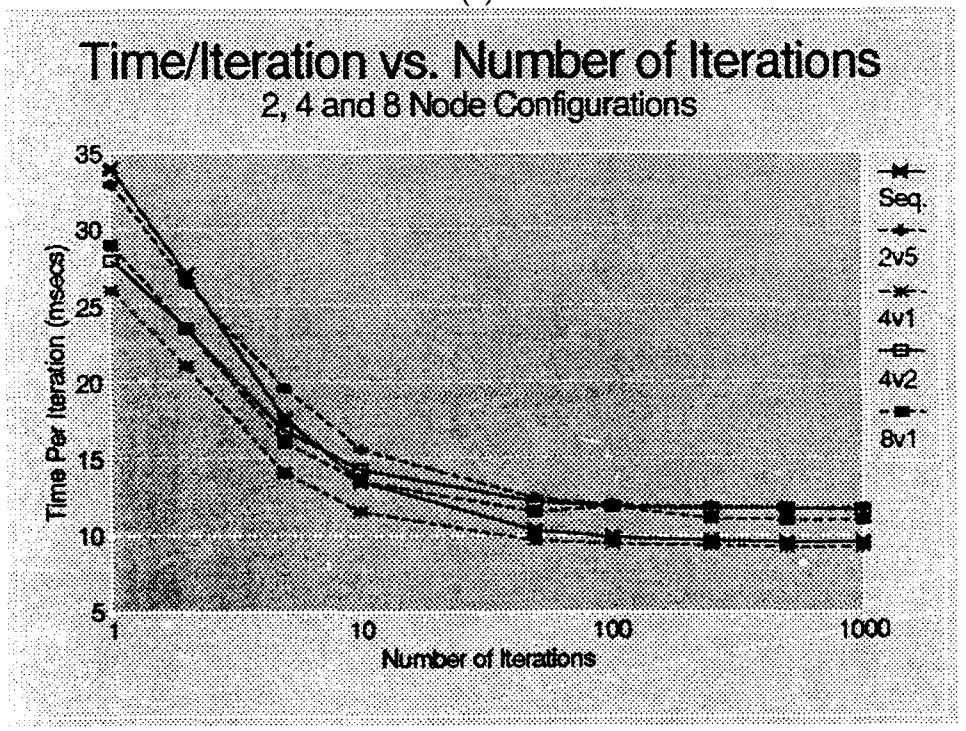
The simulated circuit in the PDESS reaches a steady state condition after two or three iterations of the simulation. This steady state condition of the PDESS circuit is reached when the states of all the objects reach a point of equilibrium. At this point the voltages, LCF, and current do not change for subsequent iterations of the simulation. When objects that provide state information to a bus object reach a steady state condition, the bus objects do not execute their floating point calculations, and for a five connection bus the time that it takes to execute the `Get_Power_Info_From` method decreases from five msecs to less than one msec. The value of T_{get} for buses decreases once a steady state condition is reached, and the amount of time that each processor spends doing computations decreases. The time that it takes to send messages between objects on separate processors, however, stays the same. Thus, there is a change in the ratio of computations versus communications. The ratio decreases when steady state conditions are reached.

Figures 26 shows the time per iteration (TPI) for the sequential simulation and the various parallel configurations. The times per iteration are based on timings of the unmodified PDESS. As can be seen in the figure, all the configurations except 1v1 and 2v2 have TPIs less than the sequential version for one and two iterations. At 10 iterations, all the configurations except 4v1 have TPIs greater than or equal to the sequential TPI. Because the TPI of the sequential simulation decreases below that of the parallel configurations, the speedups for the parallel simulations become slowdowns.[3] All the TPIs show a significant decrease as the number of iterations increase, and this decrease is due to the reduced T_{get} times for the buses.

[3] A slowdown is indicated by a speedup value S of less than one.



(a)



(b)

Figure 26. Time per Iteration for Various Object Mappings.

The parallel and the sequential simulations reach steady state conditions after the same number of iterations. The sequential simulation shows a decrease in TPI proportional to the decrease in T_{get} for buses, and a reduction in the value of T_{get} results in a decrease in the value of T_{gate}. TPI is defined as the total execution time of the simulation divided by the total number of iterations. If T_{gate} is reduced by r percent due to steady state being reached, and steady state is reached on the second iteration of the simulation (if no new inputs are applied to the objects in the simulation after the first iteration) then for the sequential simulation, TPI_{seq} can be defined by the equation:

$$\text{TPI}_{seq} = \frac{t_{gate_1} + (i-1)\left(\frac{t_{gate_1}}{r}\right)}{i} \tag{17}$$

where t_{gate_1} is the time that it takes to gate all the connections for the first iteration, and i is the number of iterations. For only one iteration $\text{TPI}_{seq} = t_{gate_1}$, but for subsequent iterations $\text{TPI}_{seq} < t_{gate_1}$. If t_{gate_1} is reduced from 35 msecs to 10 msecs due to steady state being reached after the first iteration, then plotting the equation of TPI_{seq} verses the number of iterations yields the plot shown in Figure 27, where $r = \frac{35}{10} = 3.5$. The curve of Figure 27 has the same shape as those shown in Figure 26. There is a sharp decrease in TPI initially, then the curve reaches an asymptotic limit at at around 100 iterations — where TPI is approximately equal to 10 msecs.

The parallel simulation will have a similar decrease in computations on each processor due to T_{get} decreasing. However, the amount of time spent on communications and the overhead of the logic required by the parallel implementation for gating of local and non-local connections will remain the same, even though T_{get} decreases. The time spent gating connections for the parallel simulation will be some fraction of the sequential simulation's connection gating time because only a fraction of the sequential simulation's connections will be gated on each of the parallel processor nodes. Ideally, the time spent just in computations associated with gating connections on the parallel processor nodes will be equal to the amount of time the sequential simulation spends

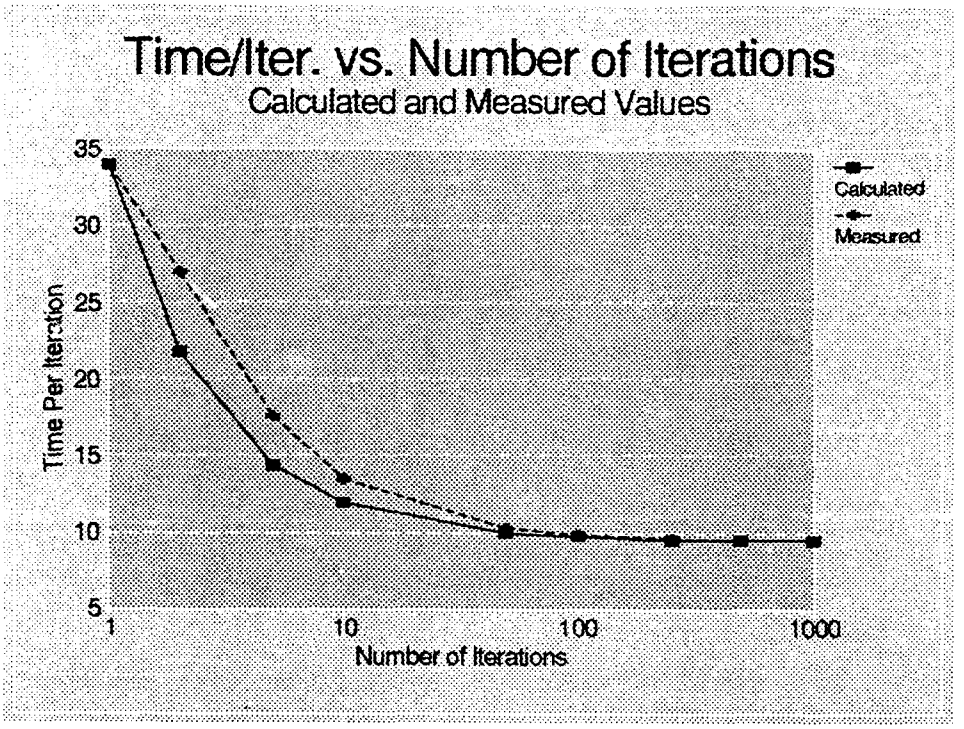


Figure 27. Calculated Time Per Iteration for the Sequential Simulation.

gating connections divided by the number of processors used in the parallel simulation. The time that the sequential simulation spends gating connections as derived from equation 17 is:

$$T_{gate_{seq}} = t_{gate_1} + (i-1)\left(\frac{t_{gate_1}}{r}\right) \tag{18}$$

and the amount of time that the parallel simulation spends in *just* the computations associated with the gating of connections is equal to:

$$T_{gate_{par}} = \frac{T_{gate_{seq}}}{p} \tag{19}$$

$$= \frac{t_{gate_1} + (i-1)\left(\frac{t_{gate_1}}{r}\right)}{p} \tag{20}$$

if the workload from the sequential simulation is divided equally among the p processors of the parallel simulation.

The parallel simulation will spend a part of its total execution time for communications and the overhead associated with extra logic of the parallel simulation. If the time that the parallel simulation spends in communications associated with the gating of non-local connections is equal to T_{comm} and the amount of overhead due to the extra logic of the parallel simulation is equal to $T_{overhead}$; then, the time per iteration of the parallel simulation is defined by the equation:

$$\mathrm{TPI}_{par} = \frac{T_{gate_{par}} + T_{comm} + T_{overhead}}{i} \tag{21}$$

$$= \frac{\frac{t_{gate_1} + (i-1)\left(\frac{t_{gate_1}}{r}\right)}{p} + T_{comm} + T_{overhead}}{i}. \tag{22}$$

The value for $T_{overhead}$ can be estimated by subtracting the execution time of the PDESS on a single node from the execution time for the sequential simulation. If the value of T_{comm} can be expressed as a percentage of $i * t_{gate_1}$ and p is equal to 2, then TPI_{par} can be plotted against the number of iterations to yield a curve as shown in Figure 28.

Each of the curves in Figure 28 marked "Par. X%" indicates the calculated values for TPI_{par} where $T_{comm} = X\%(i * t_{gate_1})$. As can be seen by the curves, the value of T_{comm} has a significant impact on the shape of the curve. Some values of T_{comm} show a speedup for a small number of iterations; but as the number of iterations increases, the sequential TPI becomes less than the parallel TPIs. This is the same trend seen in the measured TPI values for the two node configurations of the PDESS shown in Figure 26. TPI_{par} can only remain less than TPI_{seq} if the value of $T_{gate_{par}}$ meets the following criteria:

$$T_{gate_{par}} < T_{gate_{seq}} - (T_{comm} + T_{overhead}) \tag{23}$$

From the measured speedup results shown in Figure 28 it appears that configuration 4v1 is the only configuration for which this relation holds true over the range of one to 1000 iterations. Thus, 4v1 is the only configuration that maintains a speedup up to 1000 iterations. For all the other tested

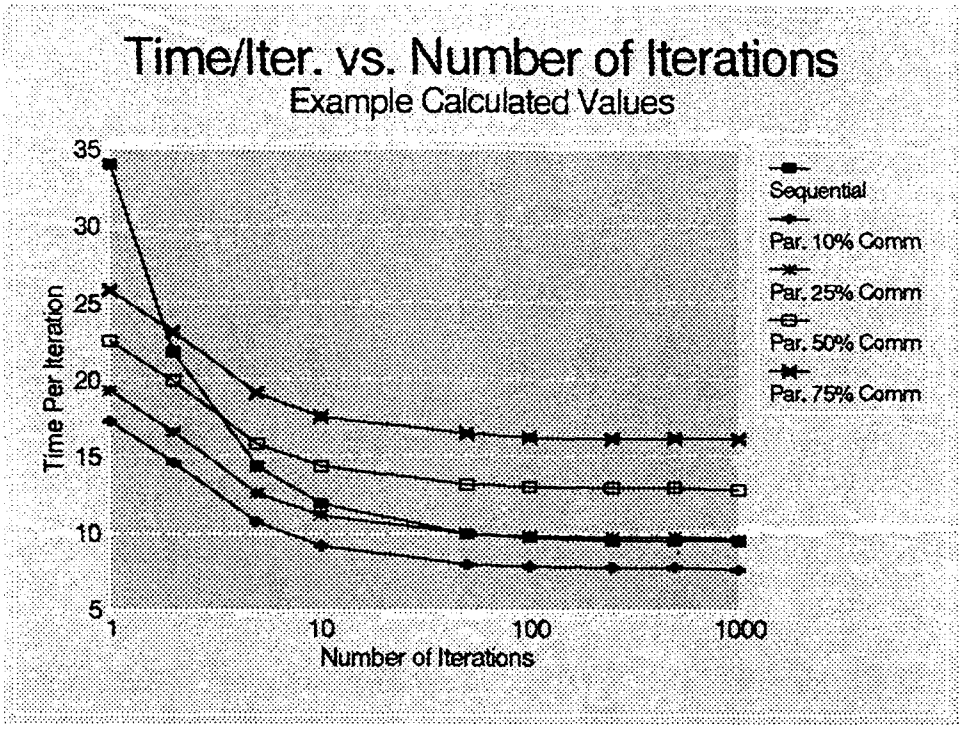


Figure 28. Calculated Time per Iteration for Parallel Simulations.

configurations, $T_{gate_{par}}$ does not maintain the relation shown in Equation 23, and thus they show a slowdown as the number of iterations increases.

The measured speedups of the PDESS show significant decreases due to the decreases in time per iteration of the simulation. The decrease in TPIs is due to the decrease in T_{get} of buses, which results in a decrease in T_{gate}. The decrease in T_{get} is due to a steady state being reached by all of the voltage, LCF and current values of the objects and the computational load of simulation tasks changing. Obviously, if the PDESS simulation is going to stay in a steady state condition for several iterations, then the parallel simulation will not show any significant speedup over the sequential version of the simulation. In fact, if the simulation stays in a steady state for long periods of time (50 to 100 iterations or more) then the simulation would probably be better handled using a discrete-event simulation. A discrete-event simulation can model the state of the DC Electrical System at the first iteration and then schedule an event to calculate the state of the system at a

time equal to the time that the 50th or 100th will occur. This avoids having to calculate all the states between the first and 50th or 100th iterations.

Lee, Rissman *et al.* state that, "Flight simulators are not event-driven. Interaction between systems in the real aircraft are continuous. Simulators model those interactions in discrete time" [30:4].[4] Thus, the state of the simulation only changing every 50 to 100 iterations may not be a realistic case for a real-time, man-in-the-loop, flight training simulator. The state of the simulation should likely change every couple of iterations due to some input such as a fluctuation of the AC Power System voltages due to changes in engine RPM, etc.. These changes in the AC Power System will feed into the DC Power System and the state of the DC Power System objects will change accordingly. Changes in object states every iteration or two of the simulation could be expected, and in this case the speedup measures done above indicate the worst case situation. In order to get an idea of the potential speedup that can be achieved if the state of the simulated objects changes every iteration of the simulation, a special series of tests are done. The next section describes these tests and their results.

6.6 Fixed T_{get} Speedup Measurements

The `Get_Power_From` method of the bus object manager was modified so that the value of T_{get} for bus objects stays the same from one iteration to the next.[5] As stated in Section 6.5.1, the value of T_{get} for buses can vary depending on the inputs received on the side of a bus. If the inputs on the side of a bus do not change between consecutive applications of the `Get_Power_From` method, then the floating point calculations for the LCF and current values of the bus are not executed when the state of the bus is read. The `Get_Power_From` method for the bus objects was modified so that the floating point calculations are *always* executed. This will keep the value of T_{get} roughly

[4] There may be some who would disagree with this view, but since this research is based on the work of Lee, Rissman *et al.* the point is not argued here.

[5] All methods are applied to bus objects using the methods provided by the `Bus_Object_Manager` package. Therefore, modifications to methods for bus objects are done in the Bus Object Manager package. See Chapter III, Section 3.5 for a description of the object manager packages.

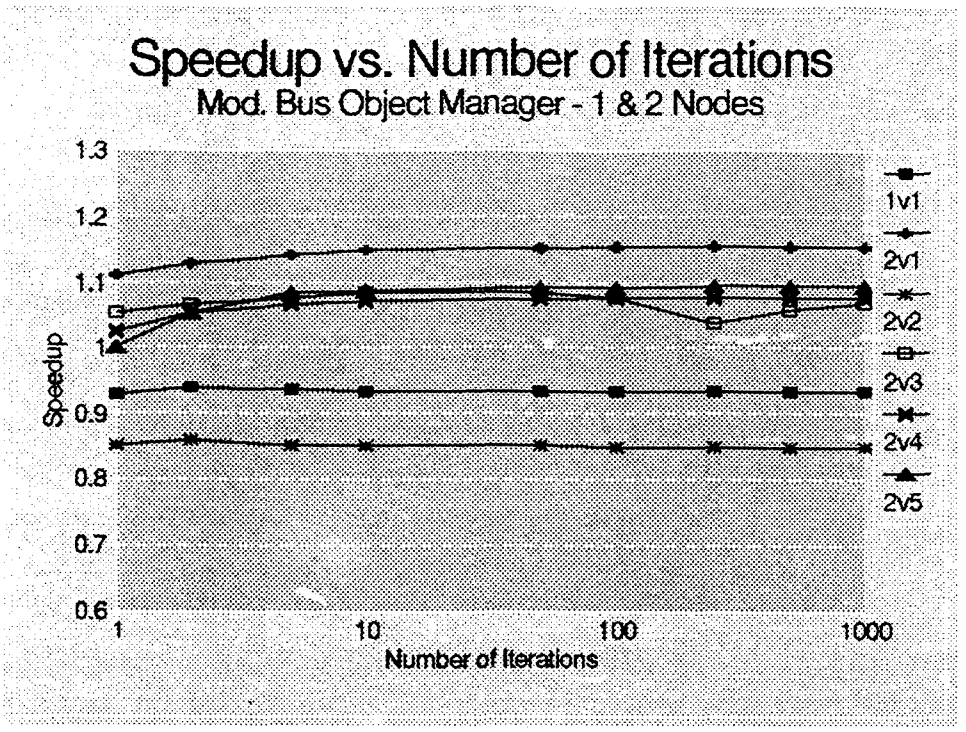
the same from one iteration of the simulation to the next even when steady state conditions are reached by the objects in the PDESS.

The speedup of the configurations shown in Figures 16 through 23 and Tables 5 and 6 were measured using the modified bus object manager, and the measured speedups are shown in the graphs in Figure 29. Figure 29(a) shows the speedup for the two node configurations, and Figure 29(b) shows the speedup for the four and eight node configurations.

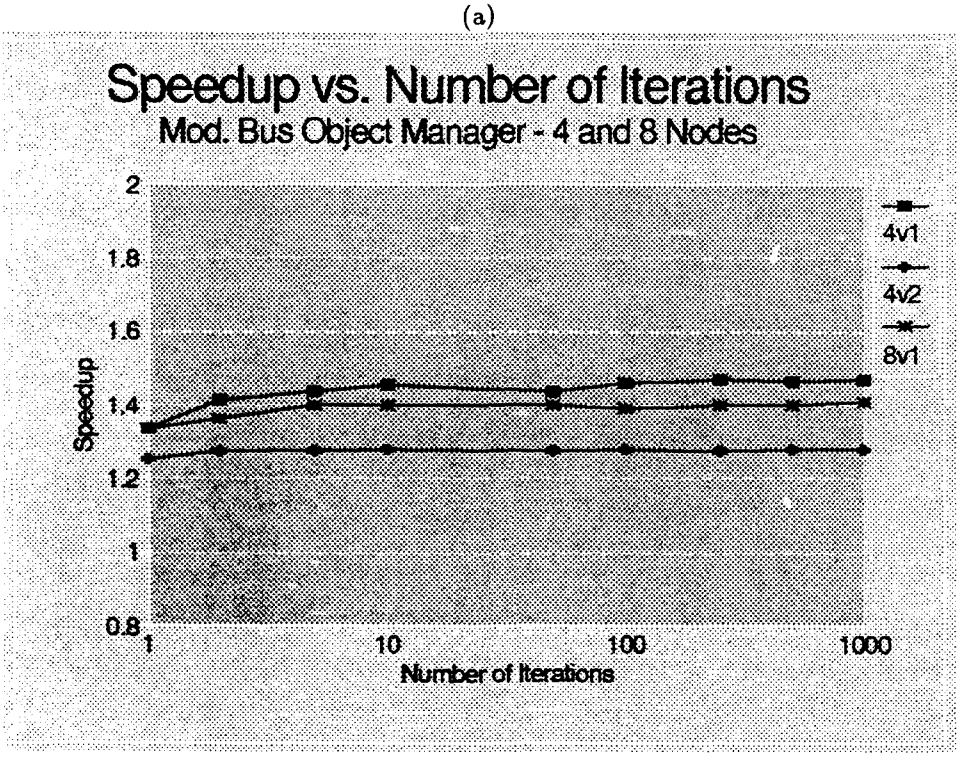
All the configurations in Figure 29, except 1v1 and 2v2, show a slight increase in speedup as the number of iterations goes from one to 10, and the speedups level off after 10 iterations. None of the speedup curves for the modified bus object manager show the same decrease seen in the original timings using the unmodified bus object manager with the variable T_{get} times.

Figure 30 shows how the speedup of the PDESS with the modified bus object manager and the speedup of the PDESS with the original bus object manager compare. The measured speedup at 1000 iterations for each configuration (1v1 through 8v1) is shown along the y-axis and the configurations are along the x-axis. The two node configurations are shown first on the x-axis in order of increasing speedup. Configuration 2v1 has the highest speedup of the two node configurations. The four node configurations are shown in order of increasing speedup after the two node configurations, and the eight node configuration is shown after the four node configurations.

Figure 30 shows that each configuration has a higher measured speedup at 1000 iterations when the modified bus object manager with the fixed T_{get} time is used. The curve for the modified bus object manager (MBOM) configurations shows that configuration 4v1 has the highest speedup (it also has the highest speedup using the unmodified bus object manager (UBOM) with variable T_{get} time). The eight node configuration 8v1 has the second highest speedup of the MBOM curve, and configuration 4v2 is the next highest. All the two node MBOM configurations except 2v2 show approximately the same speedup. Configuration 2v2 has the smallest speedup (actually a slowdown) of all the configurations for the UBOM and the MBOM. Configuration 2v2 is the only



(a)



(b)

Figure 29. Time per iteration for various object mappings using modified bus object manager with fixed T_{get}.

configuration that shows a slowdown for the MBOM configurations. The UBOM curve shows configuration 4v1 is the only configuration that has a speedup at 1000 iterations. All the other configurations show a slowdown.

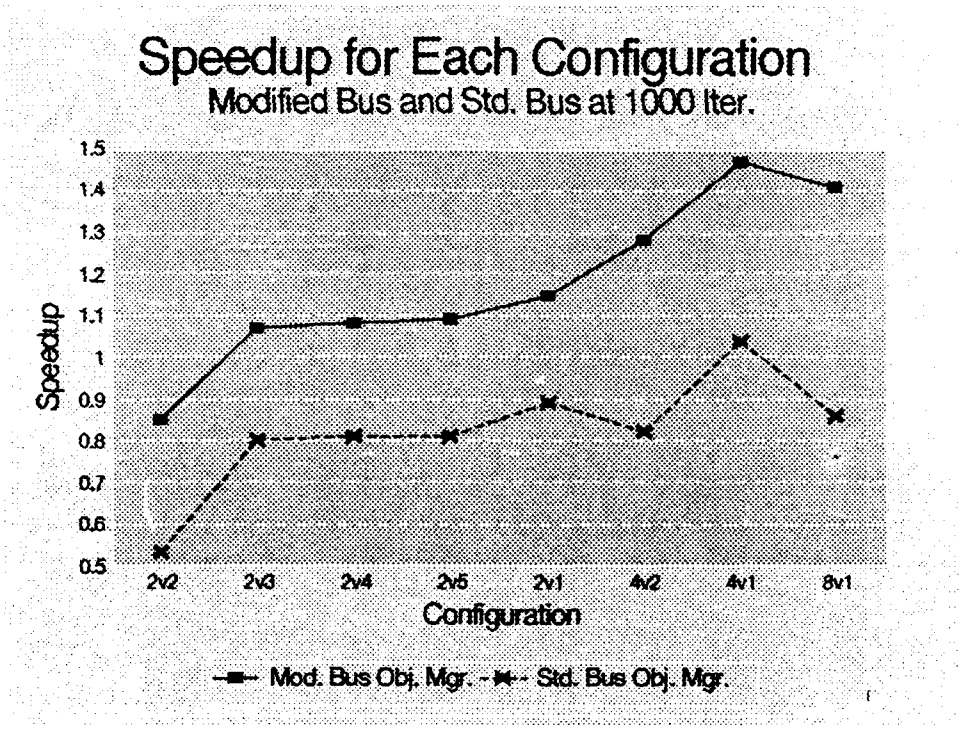


Figure 30. Measured speedup at 1000 iterations using the standard bus object manager and modified bus object manager.

6.7 Performance Results Summary

The speedup of the PDESS using the original unmodified bus object manager shows a decrease in speedup for all tested configurations as the number of iterations of the simulation is increased from one to 1000 iterations. The decrease in speedup is due to the time-per-iteration of the sequential version of the simulation decreasing to a lower value than the time-per-iteration of the parallel version.

The decrease in TPI of the sequential and the parallel simulation is caused by a decrease in T_{get} for connections involving bus objects. When the objects that provide inputs to the bus

objects reach a steady-state condition where their state outputs are not changing, the computations associated with executing a read on a bus object are significantly reduced. Therefore, T_{get} of the bus decreases, and T_{gate} of the processor decreases.

The bus object manager was modified such that T_{get} for the buses is constant. This modified version of the object manager shows the potential speedup achievable if the objects in the simulation change state every iteration or two. The measured speedups using the modified bus object manager do not show the decrease in speedup seen when the bus object managers with variable T_{get} time are used. The next chapter presents an analysis of the measured results and builds a model for explaining these results.

VII. Results Analysis of the Parallel OOD Simulation

7.1 Introduction

This chapter presents an analysis of the Parallel DC Electrical System Simulation (PDESS) for the various configurations that were tested in Chapter VI. The analysis is based on using time-line graphs and PERT (Performance and Evaluation Review Technique) networks to determine the execution time of each processor. This method of analysis allows the effects of load balancing, communications overhead and processing-order dependencies to be quantified and equations that define the behavior of the simulations to be derived. This analysis method is first presented by doing an analysis of the modified bus object manager version of configuration 2v1.

7.1.1 Analysis of Configuration 2v1. Each processor in the PDESS gates a series of connections based on the objects that are mapped to that processor.[1] Figure 31 shows how the objects from the PDESS are mapped to each of the two processors used to run configuration 2v1. This mapping of objects to processors yields 82 local connections (10 executive-level and 72 system-level connections), 2 NLSCs and 2 NLDCs for node 0. Node 1 has 40 LCs (six executive-level and 34 system-level connections), 2 NLSCs and 2 NLDCs for node 1. Since only two processors are used for configuration 2v1, the NLSCs on node 0 are the NLDCs on node 1, and the NLSCs on node 1 are the NLDCs on node 0. The NLSCs on node 0 (thus, the NLDCs on node 1) are connections 23 and 111. The NLDCs on node 0 (thus, the NLSCs on node 1) are connections 43 and 81.

Node 0, for the 2v1 configuration, gates its connections in the following manner for each iteration of the simulation:

1. Gate all executive-level connections – which are all local connections.
2. Gate local connections 7 through 20.

[1] Chapter V describes in detail how the connections are gated.

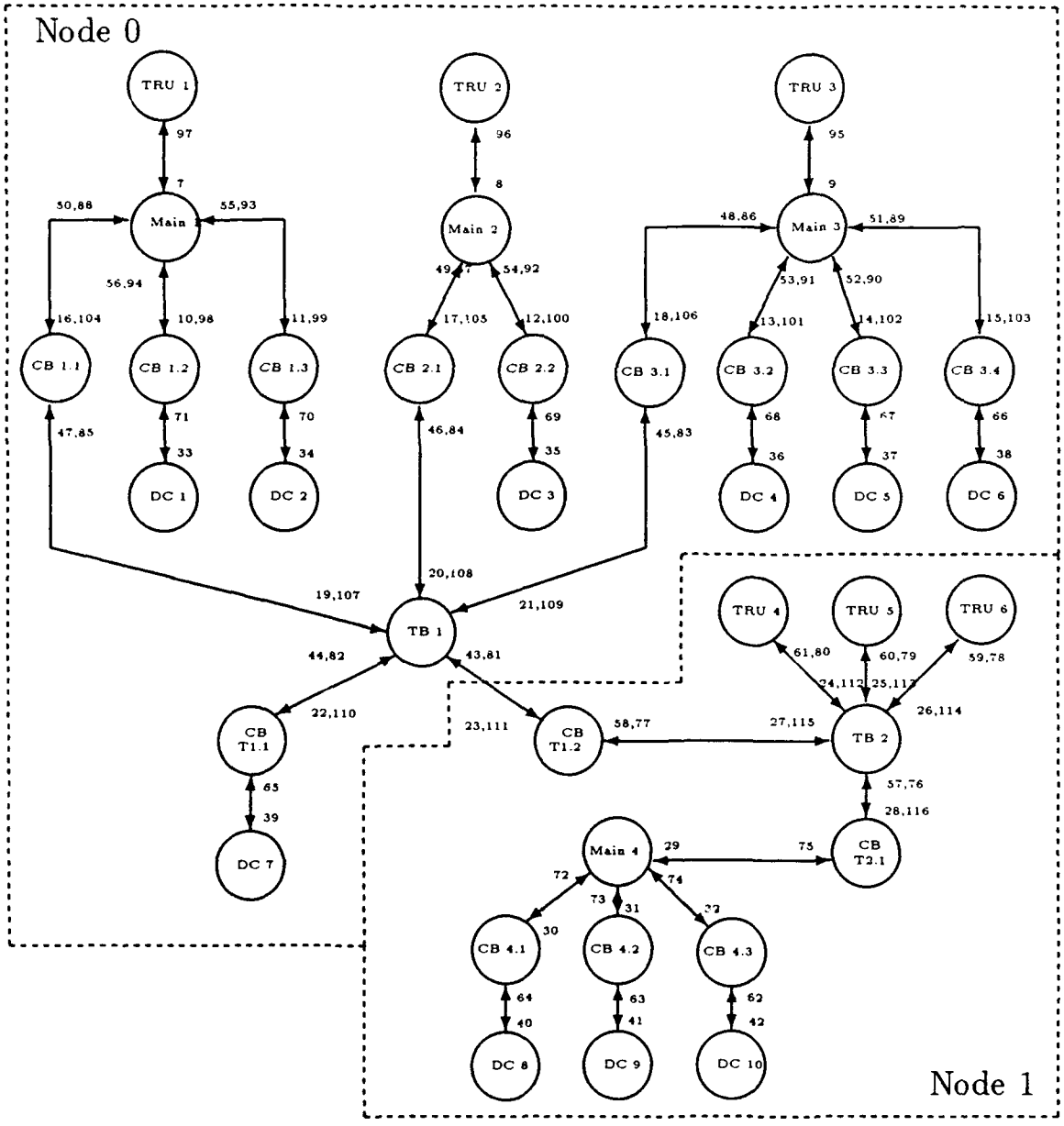


Figure 31. Mapping of the PDESS objects for configuration 2v1.

3. Gate connection 21, the first NLSC on node 0. This will send voltage and LCF data to node 1 so that node 1 can gate NLDC 21.
4. Gate local connections 33 through 39.
5. Gate connection 43, the first NLDC on node 0. If node 1 has not gated its first NLSC, connection 43, prior to the time that node 0 attempts to gate NLDC 43, then node 0 will block its connection gating until the data from node 1's NLSC 43 is received. The data sent from node 1 to node 0 is voltage and LCF data. The gating of NLDC 43 is completed by node 0 when the data from node 1 is received, read from the receive buffer and applied to the side of the Tie Bus 1.
6. After the gating of connection 43 is completed, gate local connections 44 through 56 and 65 through 71.
7. Gate NLDC 81, and block if the data from NLSC 81 on node 1 has not been received yet. Once the load/current data from NLSC 81 on node 1 is received, then complete the gating of NLDC 81.
8. Gate local connections 81 through 110.
9. Gate NLSC 111 and send load/current data to node 1.
10. Start the next iteration of the simulation by gating the executive-level connections again and repeating the connection gating steps described above.

Node 1 gates its connections in the same manner as node 0, but the connections that node 1 gates are the connections associated with the objects mapped to it. Node 1 will start the first iteration by gating NLDC 23 and blocking until the data sent from node 0's NLSC 23 is received. Local connections are then gated until NLSC 43 is gated, and then more local connections are gated until NLSC 81 is gated. After NLSC 81 is gated, NLDC 111 is gated, and the gating of NLDC 111 will be blocked until data is received from node 0's NLSC 111. After this data is received by node 1, then NLDC 111 finishes gating, and local connections 112 through 116 are gated. This completes the first iteration of the PDESS on node 1, and each iteration of the PDESS on node 1 will repeat these connection gating steps.

Figure 32 uses a time-line graph to show the connection gating described above. The arrows between time-lines indicate data being transferred from one node to another due to the gating of a NLSC. The time required to gate each series of local connections is shown in the graph, and these times were derived from measurements done using a modified version of the `DC_Power_System` package body and the modified bus object manager with fixed T_{get} time.

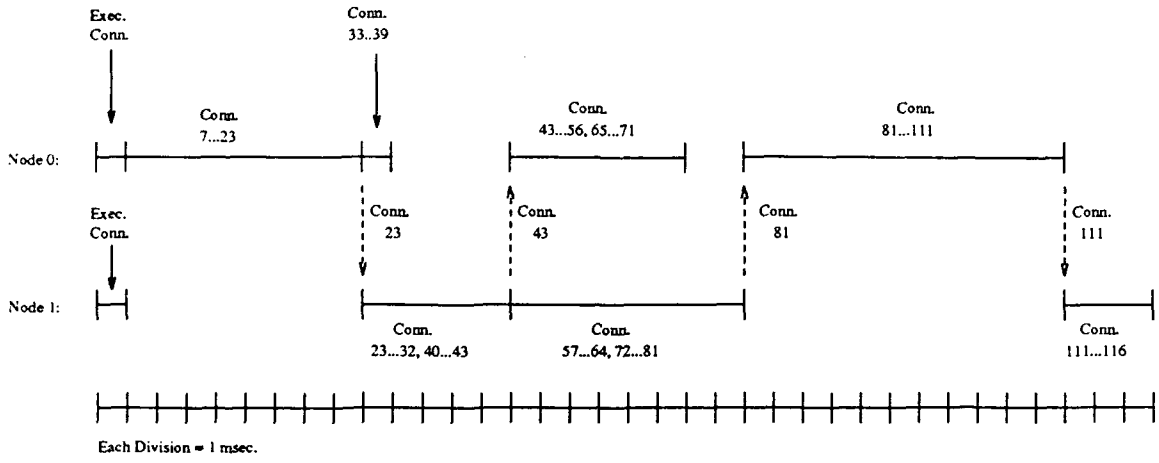


Figure 32. Configuration 2v1 connection gating time-line.

The `DC_Power_System` package body was modified to report the value of `mclock` when the PDESS starts an iteration of the simulation, ends an iteration and when a NLSC or a NLDC connection is gated. The blocking `crecvs`, that are used when an NLDC is gated, were removed from the `DC_Power_System` package so that the node processors do not block for incoming messages. Thus, the time modified `DC_Power_System` package only measures the time spent gating connections, and not the time spent blocking. This method of measuring the gating times is used since the time spent blocking by each processor can be calculated using only the gating times and the connection processing order information, and these times are the only times needed for doing the PERT analysis.

Figure 32 shows that node 0 takes 8 msecs to gate its executive-level connections, LCs 7 through 22, and NLSC 23. After NLSC 23 is gated, node 0 takes 1 msec. to gate LCs 33 through 39. Node 1 gates its executive-level connections in 1 msec. and then attempts to gate NLDC 23. Node 1 cannot complete the gating of NLDC 23 until the data from node 0 is received from NLSC 23 being gated. Thus, node 1 waits for 7 msecs (8 – 1 msecs) plus the time that it takes to transfer the voltage-LCF data from node 0 to node 1 for connection 23. The voltage-LCF data sent by NLSC 23 is a ten byte message, and from the measurements done on the iPSC/2 Hypercube it

takes approximately 0.5 msecs to transfer a message of this size.[2] (Current/load messages are eight byte messages and take approximately 0.4 msecs to transfer between nodes.) Therefore, node 1 completes the gating of NLDC 23 after waiting for 7.5 msecs.

Node 0's NLDC 43 is gated after LC 33 through 39 are gated, but NLDC 43 cannot complete gating until node 1's NLSC 43 is gated and the data received by node 0. Node 1 gates NLSC 43 five msecs after NLDC 23 is gated, so node 0 completes the gating of NLDC 43 after waiting for 4.5 msecs.[3] Once node 0 finishes gating NLDC 43, 6 msecs are used to gate LCs 43 through 56 and 65 through 71. Node 0 will then block when NLDC 81 is gated. Since node 1 takes 8 msecs to gate LCs 57 through 80, and NLSC 81, then node 0 waits for the time that it takes node 1 to gate these connections plus the time that it takes to transfer the load/current data from node 1 to node 0. As mentioned above, the time that it takes to transfer load/current data is 0.4 msecs. Thus, node 0 waits for 2.4 msecs[4] before NLDC 81 can finish gating. Node 0 then takes 11 msecs to gate LCs 81 through 110 and NLSC 111 after it receives the data from connection 81 on node 1. Node 1 attempts to gate NLDC 111 while the data from NLSC 81 is being transferred between node 1 and node 0, and it cannot complete the gating of NLDC 111 until node 0 has gated NLSC 111 and the data from node 0 has been received by node 1. Thus, node 1 waits 0.4 msecs while the data from NLSC 81 is being sent to node 0, and node 0 waits 11 msecs plus the time that it takes to transfer the data for connection 111 from node 0 to node 1 – which is 0.4 msecs. Node 1 waits a total of 11.8 msecs[5] before it can complete the gating of NLDC 111, then it takes 3 msecs to gate LCs 112 through 116.

[2] See Chapter IV, Table 3.
[3] 5 + 0.5 – 1 = 4.5 msecs.
[4] 8 + 0.4 – 6 = 2.4 msecs.
[5] 0.4 + 11 + 0.4 = 11.8 msecs.

Summing the time spent gating connections and the time spent waiting to gate connections yields the following execution times for the node 0 and for node 1,

$$T_{e_0} = 8 + 1 + 4.5 + 6 + 2.4 + 11 \text{ msecs} \quad (24)$$

$$= 32.9 \text{ msecs} \quad (25)$$

$$T_{e_1} = 1 + 7.5 + 5 + 8 + 11.8 + 3 \text{ msecs} \quad (26)$$

$$= 36.3 \text{ msecs} \quad (27)$$

The execution time of the parallel simulation, $T_{e_{par}}$, is the maximum of the processor execution times. Thus, $T_{e_{par}}$ for one iteration of the PDESS using the modified bus object manager for configuration 2v1 is equal to,

$$T_{e_{par}} = Max(T_{e_0}, T_{e_1}) \quad (28)$$

$$= 36.3 \text{ msecs} \quad (29)$$

The actual measured time of this configuration 2v1 using the modified bus object manager with fixed T_{get} is 36 msecs for one iteration. Thus, the calculated execution time for one iteration is within 0.8% of the measured execution time.

The time-line graph in Figure 32 provides some insight into how the connections processing dependencies affect the execution times of each of the processors during a single iteration of the PDESS. It can be seen that node 0 spends 26 msecs gating connections and 6.9 msecs blocking for data from node 1. Node 1 spends 17 msecs gating connections and 19.3 msecs blocking for data from node 0.

Another method of determining the execution time of the PDESS is by using a Performance and Evaluation Review Technique (PERT) network [42] [18] and determining the critical path

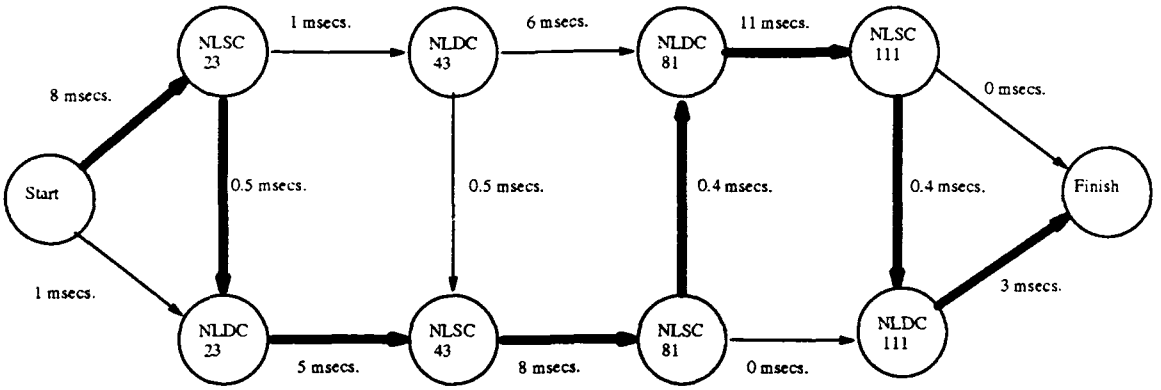


Figure 33. Configuration 2v1 connection gating PERT network.

through the network. Using the PERT network and finding the critical path avoids having to calculate the waiting times for each of the processors as was done above. A critical path for configuration 2v1 is determined by using the processing order dependencies information and the connection processing times of each processor shown in Figure 32, and generating a PERT network as shown in Figure 33. By finding the critical path from the start node to the finish node, the execution time of one iteration of the parallel simulation can be determined.

The nodes shown in Figure 33 consist of a start node, a finish node and all the NLSCs and NLDCs executed by each node in a single iteration of the PDESS simulation for configuration 2v1. The first node in Figure 33 is the start node. This represents the point at which node 0 and node 1 start gating connections for the first iteration of the simulation. Node NLSC 23 is the first non-local connection gated by node 0, and the 8 msec time shown from the start node to node NLSC 23 is the time that it takes node 0 to gate its executive-level connections, the local connections before NLSC 23, and NLSC 23. NLDC 23 is the first non-local connection that is gated by node 1, and the 1 msec time shown between the start node to node NLDC 23 is the time that it takes node 1 to gate its executive level connections and gate NLDC 23. The 0.5 msec time shown between nodes NLSC 23 and NLDC 23 is the time that it takes to transfer voltage-LCF data between nodes 0 and 1 since connection 23 transfers voltage-LCF. The 0.4 msec time shown between nodes NLSC 81 and NLDC 81 is the time that it takes to transfer load/current data between nodes since connection

81 transfers load/current information. The one msec time between NLSC 23 and NLDC 43 is the time that it takes node 0 to gate all the local-connections between NLSC 23 and NLDC 43. The five msec time between NLDC 23 and NLSC 43 is the time that it takes node 1 to gate the local-connections between NLDC 23 and NLSC 43. The times between the other NLSC and NLDC nodes are determined from Figure 32 in this same manner. The time between NLSC 111 and the finish node is the time that it takes node 0 to gate the local connections after NLSC 111, but NLSC is the last connection gated by node 0 during a single iteration of the PDESS. If a second iteration had been started then node 0 would have started gating the executive-level connections and the local system-level connections leading up to NLSC 23 and NLDC 43. (A PERT network analysis for two iterations of the simulation will be done later in this section.) The time from NLDC 111 to the finish node is the time that it takes node 1 to finish gating the last local connections on node 1 – connections 112 through 116.

The execution time of a single iteration of the simulation is equal to the length of the *critical path* for the PERT network shown in Figure 33.[6] The critical path length (or time) is the length of the longest path from the start node through the network to the finish node. The dark arrows in Figure 33 indicate the critical path through this network. The length of the critical path through the PERT network is equal to,

$$8 + 0.5 + 5 + 8 + 0.4 + 11 + 0.4 + 3 = 36.3 \text{ msecs}$$

which is the same execution time, $T_{e_{par}}$, calculated using the the time-line graph shown in Figure 32. Using the PERT network and finding the critical path through the network allows calculating $T_{e_{par}}$ without having to calculate the waiting time or the execution time of each of the processors, but these values can be calculated from the network if needed. The PERT network is a more concise representation of the execution time dependencies between each of the processors. The PERT

[6] Please refer to [42] or [18] for a description of the technique used to determine the critical path through a PERT network.

network can be built from measurements of the time required to process connections between NLSCs and NLDCs, and from the measurements of the time from the start of an iteration to the time that the first NLSC or NLDC is gated and the time required to complete and iteration after the last NLSC and NLDC are gated.

Figure 34 shows the PERT network for two iterations of the PDESS for configuration 2v1 using the modified bus object manager with fixed T_{get}. The critical path through the network is indicated by the darker edges of the network and the critical path has a length of 69.2 msecs, so the predicted value for $T_{e_{par}}(2) = 69.2$ msecs. The actual measured time for two iterations of the PDESS for configuration 2v1 is 70 msecs. Thus, the predicted $T_{e_{par}}(2)$ and the actual measured time are very close.

An interesting observation about the execution time of the PDESS for configuration 2v1 can be made – the value of $T_{e_{par}}(2) \neq 2 * T_{e_{par}}(1) = 72.6$ msecs. Due to the fact that node 0 starts on the second iteration while node 1 is finishing the first, the execution time of the second iteration is equal to the time of the first iteration minus the time that it takes node 1 to get the data from node 0 for NLDC 111 minus the time required to finish gating local connections 112 through 116. The time that it takes node 1 to receive the data from node 0 for connection 111 is 0.4 msecs and the time that it takes for node 1 to finish gating the last local connections is 3 msecs. Thus, the time that it takes for two iterations is

$$T_{e_{par}}(2) = T_{e_{par}}(1) + (T_{e_{par}}(1) - 3.4) \text{ msecs} \quad (30)$$

$$= 36.3 + 32.9 \text{ msecs} \quad (31)$$

$$= 69.2 \text{ msecs.} \quad (32)$$

Each iteration of the simulation after the first iteration will take 32.9 msecs due to node 0 starting iteration $i + 1$ in 3.4 msecs before node 1 starts iteration $i + 1$, and the following equation can be

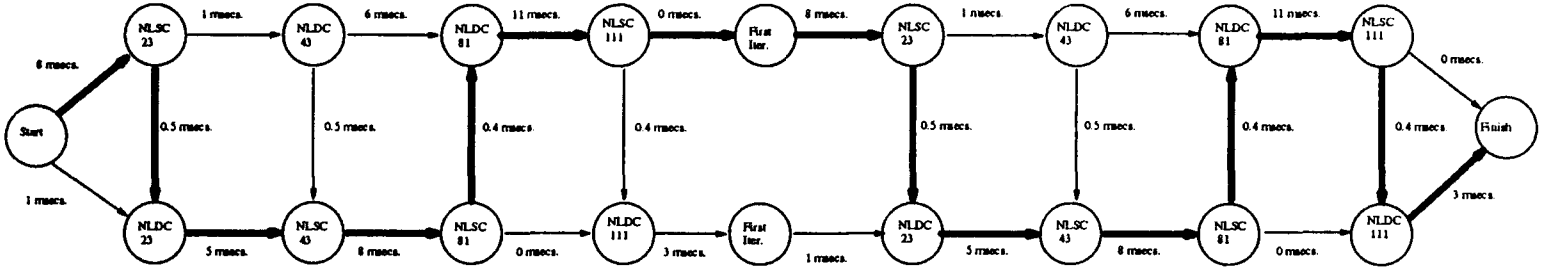


Figure 34. Configuration 2v1 connection gating PERT network for two iterations.

Table 7. Calculated and Measured Execution Times for Configuration 2v1 with fixed T_{get}.

	$T_{epar}(i)$		
Iterations	Calculated	Measured	%Difference
1	36	36	1
2	69	70	-1
5	168	170	-1
10	332	336	-1
50	1648	1665	-1
100	3293	3325	-1
250	8228	8310	-1
500	16453	16600	-1
1000	32903	33232	-1

used to predict the execution time of the PDESS for configuration 2v1:

$$T_{epar}(i) = T_{epar}(1) + (i-1)(T_{epar}(1) - 3.4) \text{ msecs} \quad (33)$$

$$= 36.3 + 32.9(i-1) \text{ msecs.} \quad (34)$$

where i is the number of iterations of the simulation.

Table 7 shows the measured execution times of the PDESS for configuration 2v1, the execution time calculated using Equation 33, and the percentage difference between the two values. The percentage difference of the values are determined by using the following equation:

$$\%\text{diff} = \frac{\text{Calculated } T_{epar}(i) - \text{Measured } T_{epar}(i)}{\text{Measured } T_{epar}(i)}. \quad (35)$$

As can be seen by the small percentage differences of the measured and the calculated execution times (a maximum %difference of 1.2%), Equation 33 shows a good correlation with the measured performance results. Given the measured execution times of the sequential simulation and Equation 33, then the speedup of the PDESS for configuration 2v1 with fixed T_{get} time can be predicted within 1.2% of the actual values.

Table 8. Calculated and Measured Execution Times for Configuration 2v3 with fixed T_{get}.

	$T_{e_{par}}(i)$		
Iterations	Calculated	Measured	%Difference
1	37	38	-2
2	73	74	-1
5	181	180	0
10	360	356	1
50	1791	2158	-17
100	3581	3558	1
250	8952	9247	-3
500	17902	18113	-1
1000	35802	35849	0

Applying the PERT technique described above for configurations 2v3, 2v4, and 4v1, and deriving equations based on the $T_{e_{par}}(1)$ and $T_{e_{par}}(2)$ yields the equations for $T_{e_{par}}(i)$:[7]

$$\text{Config. 2v3:} \quad T_{e_{par}}(i) = 37.3 + 35.8(i-1) \text{ msecs} \tag{36}$$

$$\text{Config. 2v4:} \quad T_{e_{par}}(i) = 38.8 + 36.2(i-1) \text{ msecs} \tag{37}$$

$$\text{Config. 4v1:} \quad T_{e_{par}}(i) = 29.6 + 26.2(i-1) \text{ msecs} \tag{38}$$

A comparison of the calculated and the measured execution times for each of these configurations is shown in Tables 8, tab:2v4-meas-v-calc and tab:4v1-meas-v-calc. Comparing the calculated and measured execution times shows a %difference of less than 3% or less for all the execution times calculated using the above equations, except for the configuration 2v3 at 50 iterations. This deviation may be due to an external factor affecting the measured execution time of simulation when configuration 2v3 was tested at 50 iterations.[8]

[7] Equations for configurations 2v2, 2v5, 4v2 and 8v1 were not derived due to limited testing time and publication deadlines. The PERT analysis technique should generate the same accuracy of results for these configurations.

[8] Increases in execution time of the PDESS occurred on occassions when other users ran programs on nodes of the hypercube not allocated to testing the simulation. The deviation in the time for configuration 2v3 is likely due to such an effect.

Table 9. Calculated and Measured Execution Times for Configuration 2v4 with fixed T_{get}.

	$T_{e_{par}}(i)$		
Iterations	Calculated	Measured	%Difference
1	39	39	-1
2	75	75	0
5	184	182	1
10	365	360	1
50	1813	1785	2
100	3623	3565	2
250	9053	8909	2
500	18103	17812	2
1000	36203	35614	2

Table 10. Calculated and Measured Execution Times for Configuration 4v1 with fixed T_{get}.

	$T_{e_{par}}(i)$		
Iterations	Calculated	Measured	%Difference
1	30	30	-1
2	56	56	0
5	134	135	0
10	265	265	0
50	1313	1337	-2
100	2623	2626	0
250	6553	6526	0
500	13103	13086	0
1000	26203	26092	0

7.2 *Performance Analysis Summary*

The behavior of the PDESS (using the modified bus object manager) can be analyzed using a PERT network and measurements of the time required to gate connections between NLSCs and NLDCs from a single iteration of the PDESS simulation. A measurement of the amount of time spent blocking by a processor is not required in order to use the PERT network analysis method.

Using the results from the PERT network analysis, equations for the execution time of several versions of the PDESS were derived. These equations showed an accuracy of 5% or better for calculating the execution time of the tested configurations over the range of one to 1000 iterations. These equations for the execution time of the PDESS configurations can be used to determine the potential speedup of the various configurations as a function of the number of iterations of the simulation.

VIII. Conclusions and Recommendations

8.1 Summary of Research Effort

This research effort addressed modifying the design of the DC Electrical System Simulation (DESS) to allow a parallel implementation of the DESS in order to reduce its execution time. The DESS was designed using the SEI's OOD Paradigm for Flight Simulators and the design was modified to allow the DESS to be implemented on an iPSC/2 Hypercube computer - a distributed memory MIMD computer. The modifications to the original design were limited to only adding extensions to the original design to allow the medium-grain parallelism in the DESS design to be exploited and implemented on the hypercube. Most of the original design (and code) was left unmodified because the SEI's Paradigm is designed with the concept of being implemented on a parallel computer system [30], and this research attempted to provide an example of how a design based on the SEI Paradigm can be implemented on a parallel computer.

8.2 Conclusions

8.2.1 Maximum Speedup. The maximum speedup of a particular configuration of the PDESS is limited by the length of the critical path in its connection gating PERT network as was shown in Chapter VII. Reducing the length of the critical path for a given configuration should increase the speedup achieved.

8.2.2 Impact of Variable Workload. Varying workloads can significantly impact the speedup achieved for any configuration of the PDESS. This was demonstrated in Chapter VI.

8.2.3 Adding Concurrency to the SEI Paradigm. The SEI OOD Paradigm for Flight Simulators can support medium-grain concurrency at the object and connection-gating level. This was demonstrated in the design and implementation of the PDESS, and in the concurrency analysis

in Chapter IV. Modifications are required in the design and implementation of "systems" and connections in order to support this level of concurrency.

8.2.4 "Object-based" Paradigm. The SEI OOD Paradigm is not a "true" object-oriented design method, but it provides a good structured method of implementing a flight simulation using an "object-based" design. The paradigm is not truly object-oriented because several objects are grouped into a single object by the paradigm. In other words, the paradigm does not generate as many objects as would be generated by applying an object-oriented design methodology such as that presented by Booch [8]. This difference in design is due to the design goal of the paradigm to reduce the nesting of objects, and the design method presented in the paradigm meets this goal.

8.3 Summary of Contributions

8.3.1 Extensions for Parallel Design. The design of the DC Electrical System Simulation was modified to provide a parallel design by introducing the following design extensions to the SEI Paradigm:

- Local and non-local source and destination connections. The local connections are connections between objects on the same processor, and non-local connections are connections between objects on different processors.
- "Systems" were modified to support the gating of local and non-local connections using a single **gate** procedure, and a topological list method is used to allow the gating of connections such that the connection processing-order dependencies are handled properly. The topological list is a linear list of connections that can be processed in order from the first connection in the list to the last, and all the connection processing order dependencies will be met. When a non-local connection is gated and the source for the connection is on another processor, then the gating of the connections is blocked until the source data is received from the processor

with the source connection. By blocking the connection processing until the source data is received, the pro·essing-order dependencies are maintained.

- System Aggregates were modified to support system objects being distributed over several processors. In the original design, a system aggregate instantiates all the objects in a system and provides named access to all the objects that are part of the system. In the parallel design, a system aggregate is on each processor and it only instantiates those objects that are mapped to that processor for that system. It provides location information for objects that are instantiated on other processors.
- The design of the "executive" was not modified in the parallel design, but the executive connections were modified to support local and non-local connections. A copy of the executive is instantiated on each processor in the parallel system.

The parallel implementation of the DC Electrical System Simulation provides an example of implementing a design using these design extensions. These design extensions are changes that can be applied to other simulations designed using the SEI OOD Paradigm to derive a parallel design.

8.3.2 Performance Analysis Technique. This research showed how PERT networks can be used to describe and analyze the performance of a parallel simulation design using the SEI Paradigm and the extensions noted above. The PERT network provides a way to analyze the performance of a given object mapping and to model the performance. If the connection gating time between non-local connections can be determined prior to implementing the parallel design, it may be possible to gain some insight into the potential speedup that can be gained by parallelizing the simulation.

8.3.3 Performance Considerations. The effect of changes in computational workload was demonstrated and analyzed for the parallel DC Electrical System (PDESS). As steady state conditions were reached by the objects in the PDESS, the value of T_{get} of buses decreased and the execution time per iteration (TPI) of the simulation decreased. The sequential simulation showed

a 350% decrease in execution TPI due to T_{get} decreasing, but the parallel simulation showed less of a decrease in execution TPI. Thus, after five or more iterations the parallel simulation showed a slowdown instead of a speedup due to a smaller decrease in TPI than the sequential simulation. The smaller decrease in TPI of the parallel simulation was due to communication overhead staying the same when computational time was decreasing in proportion to T_{get} decreasing. When T_{get} for the buses was held constant, then a consistent speedup (or slowdown) was measured for most all of the tested configurations.

The change from a speedup to a slowdown for the PDESS demonstrates the effects of changes in computation workload on performance. These changes in computational workload can adversely affect speedup – as was demonstrated in the tests of the PDESS using the unmodified bus object manager with variable T_{get}.

8.4 Recommendations for Further Research

A method of extending the SEI's OOD Paradigm to support the design and implementation of a single flight simulation subsystem was demonstrated and tested. The following topics are recommendations for future research:

- Determine if the PERT analysis method presented in Chapter VII can be used to predict the expected performance of a simulation based on data from a sequential version of the simulation. If the PERT method can be used to predict the potential speedup that can be achieved by parallelizing a particular simulation, then a determination can be made as to whether the potential increase in performance is worth the effort of parallelizing the simulation.
- Modify the PDESS to implement a circuit model that has a higher level of parallelism in the basic simulation algorithm. The circuit model implemented by the SEI in the original design has complex connection gating dependencies, and these dependencies limit the potential

speedup that can be gained. If a simulation model for the circuit can be designed that has more parallelism, then the potential speedup will be higher.

- Determine and characterize the effects on speedup for changes in the ratio of computational workload to communications overhead (computational grain size) for a parallel simulation built using the method presented in this research. This can provide insight into what computational grain sizes are appropriate to achieve a speedup for the parallel designs that can be developed using the SEI paradigm and the extensions presented in this research.
- Implement and test the PDESS design on a shared memory architecture. This research could determine if the design extensions used for implementing the PDESS can be applied to a shared memory architecture, or if the extensions required for implementing a parallel design on a shared memory machine are different from those presented in this research.
- Use the SEI OOD Paradigm as the basis for building a discrete-event simulation paradigm for sequential and distributed OOD simulations. Gating of connections could be the events that are scheduled. The research should focus on what extensions are required to the paradigm to support events and scheduling of events.

Appendix A. *Building the DESS and the PDESS*

A.1 PDESS Build File

Two executable files are required to run the Parallel DC Electrical System Simulation on the AFIT iPSC/2 Hypercube. These programs are:

- The simulation program that runs on the iPSC/2 host, `host_main`.
- The simulation program that runs on the iPSC/2 nodes, `Test_sim.a`.

The following is a listing of a batch file that can be used to build the parallel DC Electrical System code:

```
a.mklib
a.path -a /usr/ipsc/ada/lib
ada VCHARUTIL.a      # Spec. for the character utilities package
ada VSTRINGTL.a      # Spec. for the string utilites package
ada global_.a        # Spec. for the global types
ada el_.a            # Spec. for the electrical units
ada flt_names_.a     # Spec. for the flight system names
ada bus_.a           # Spec. for the Bus Object Manager
ada cb_.a            # Spec. for the Circuit Breaker Object Manager
ada tru_.a           # Spec. for the TRU Object Manager
ada dc_.a            # Spec. for the DC Power System
ada ac_agg_host_.a   # Spec. for host version of AC Power System Agg.
ada dc_agg_host_.a   # Spec. for host version of DC Power System Agg.
ada ds_agg_host_.a   # Spec. for host version of Dummy System Power System Agg.
ada ac_agg_.a        # Spec. for node version of AC Power System Agg.
ada dc_agg_.a        # Spec. for node version of DC Power System Agg.
ada ds_agg_.a        # Spec. for node version of Dummy System Power System Agg.
ada cb_link_.a       # Spec. for the Circuit Breaker Linkage Package
ada fl_conn_.a       # Spec. for Flight Executive Connections
ada fl_.a            # Spec. for Flight Executive
ada BCHARUTIL.a      # Body for the character utilities package
ada BSTRINGTL.a      # Body for the string utilities package
ada tru.a            # Body for TRU Object Manager
ada bus.a            # Body for Bus Object Manager
ada cb.a             # Body for Circuit Breaker Object Manager
ada cb_link.a        # Body for Circuit Breaker Linkage package
ada ac_agg.a         # Body for node version of AC Power System Agg.
ada dc_agg.a         # Body for node version of DC Power System Agg.
ada ds_agg.a         # Body for node version of Dummy System Power System Agg.
```

```
ada dc.a            # Body for DC Power System
ada fl_conn.a       # Body for Flight Executive Connections
ada fl.a            # Body for Flight Executive

echo "Making the host_sim.a HOST simulation program..."
ada -o host_sim -lhost -lsocket -M host_sim.a

echo "Making the Test_sim.a NODE simulation program..."
ada -o test_sim -lnode -M Test_sim.a
```

Executing the above batch file will build a version of the PDESS that has the bus object manager with variable T_{get} times. Compile and link the program **long_bus.a** with the node program using the following commands to build a version of the PDESS with fixed T_{get} time for the bus object manager:

```
ada long_bus.a
a.ld test_sim -lnode -o test_sim
```

A.2 Running the PDESS Simulation

The PDESS simulation is started by entering the executable file name, **test_sim**, at the UNIX prompt. The program will prompt the user for all required information, and will read object mapping information from the files **ac_cfg.ext**, **dc_cfg.ext** and **ds_cfg.ext**. The filename extension, **.ext**, is entered by the user at run time and the object mapping files with this extension should be created prior to running the program.

The state of all the objects in the simulation can be printed using the "print state of objects" menu option. The number of iterations to be executed by the simulation is specified by the user, or the user can have the simulation automatically run a series of test runs for increasing numbers of iterations from 1 to 1000. Also, the user may specify how many times the simulation is to be run using the same number of iterations, e.g., the user can specify that the simulation be run for 500 iterations three times. This feature is provided to allow running the same number of iterations to determine any variation in timing results.

Timing results can be affected by more than one user executing programs on separate parts of the AFIT iPSC/2 Hypercube. All the results reported in this research were collected with the PDESS being the only program executing on the hypercube when two or more nodes were used for testing. No other programs were running on any unused nodes.

A.3 DESS Build File

The following batch file builds the sequential version of the DC Electrical System Simulation that executes on a single node the iPSC/2 Hypercube.

```
a.mklib
a.path -a /usr/ipsc/ada/lib

ada VCHARUTIL.a       # Spec. of character_utilities
ada BCHARUTIL.a       # Body of character_utilities
ada VSTRINGTL.a       # Spec. of string_utilities
ada BSTRINGTL.a       # Body of string_utilities
ada comm_globals.a    # Spec. of comm_globals

echo "Making the host_main.a HOST program for seq. simulation program..."
ada -o host_main -lhost -lsocket -M host_main.a

ada VCHARUTIL.a       # Spec. of character_utilities
ada BCHARUTIL.a       # Body of character_utilities
ada dc_.a             # Spec. of dc_power_system
ada VSTRINGTL.a       # Spec. of string_utilities
ada BSTRINGTL.a       # Body of string_utilities
ada el_.a             # Spec. of electrical_units
ada tru_.a            # Spec. of tru_object_manager
ada tru.a             # Body of tru_object_manager
ada fl_conn_.a        # Spec. of flight_executive_connections
ada comm_globals.a    # Spec. of comm_globals
ada flt_names_.a      # Spec. of flight_system_names
ada bus_.a            # Spec. of bus_object_manager
ada long_bus.a        # Body of bus_object_manager
ada global_.a         # Spec. of global_types
ada ac_agg_.a         # Spec. of ac_power_system_aggregate
ada cb_.a             # Spec. of cb_object_manager
ada cb.a              # Body of cb_object_manager
ada dummy_agg_.a      # Spec. of dummy_system_aggregate
ada dc_agg_.a         # Spec. of dc_power_system_aggregate
ada cb_link_.a        # Spec. of cb_linkage_interface
ada cb_link.a         # Body of cb_linkage_interface
```

```
ada fl_conn.a          # Body of flight_executive_connections
ada dc.a               # Body of dc_power_system
ada fl_.a              # Spec. of flight_executive
ada fl.a               # Body of flight_executive

echo "Making the node_main.a NODE program for seq. simulation program..."
ada -o node_main -lnode -M node_main.a
```

A.4 Running the PDESS Simulation

The DESS program created using the batch file shown above is executed by entering the executable file name, **host_main**. The menu interface is the same as that used for the parallel version of the simulation.

Bibliography

1. Akl, Selim G. *The Design and Analysis of Parallel Algorithms*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc, 1989.
2. Baezner, Dirk, et al. "Sim++: The Transition to Distributed Simulation." *Proceedings of the SCS Multiconference on Distributed Simulation22*. Simulation Series. 211–218. 1990.
3. Bain, William L. "A Global Object Name Space for the Intel Hypercube." *The Third Conference on Hypercube Concurrent Computers and Applications1*. 562–567. January 1988.
4. Bain, William L. and Shala Arshi. "Hypersim: A Hypercube Simulator for Parallel Systems Performance Modeling." *The Third Conference on Hypercube Concurrent Computers and Applications1*. 792–799. January 1988.
5. Beckman, Brian, et al. "Distributed Simulation and Time Warp Part 1: Design of Colliding Pucks." *Proceedings of the SCS Multiconference on Distributed Simulation19*. Simulation Series. 56–60. 1988.
6. Bensley, E. H., et al. *Distributed Object Oriented Programming*. Technical Report RADC-TR-89-339, The MITRE Corporation, February 1990 (AD-A219 689).
7. Booch, Grady. *Software Components with Ada*. The Benjamin/Cummings Publishing Company, Inc., 1987.
8. Booch, Grady. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1991.
9. Chandy, K. M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, *24*:198–206 (April 1981).
10. Chandy, K. Mani and Jayadev Misra. *Parallel Program Design: A Foundation*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc, 1989.
11. Cohen, Neil and Joseph Reynolds. "System Test Environment: A Real-Time, Man-In-The-Loop Fleet Simulator to Suppport Testing of Development Equipment." *Proceedings of the SCS Multiconference on Object Oriented Simulation*, edited by Antonio Guasch. 23–37. San Diego, California: Simulation Councils, Inc., 1990.
12. Cohen, Norman H. *Ada as a Second Language*. McGraw-Hill Book Company, 1986.
13. Corbin, M.J. and G. F. Butler. "A Toolkit for Object-Oriented Simulation in Ada." *Proceedings of the SCS Multiconference on Object Oriented Simulation*, edited by Antonio Guasch. 13–18. San Diego, California: Simulation Councils, Inc., 1990.
14. DeCegama, Angel L. *The Technology of Parallel Processing: Parallel Processing Architectures and VLSI Hardware*, *1*. Prentice-Hall, Inc., 1989.
15. Deitel, Harvey M. *An Introduction to Operating Systems*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc, 1990.
16. Department of Defense. *The Department of Defense Critical Technologies Plan for the Committees on the Armed Services United States Congress*. Technical Report AD-A219 300. March 1990.
17. Doyle, Robert J. "Object-Oriented Simulation Programming." *Proceedings of the SCS Multiconference on Object Oriented Simulation*, edited by Antonio Guasch. 1–6. San Diego, California: Simulation Councils, Inc., 1990.

18. Eris, Rene L. and Bruce N. Backer. *An Introduction to PERT-CPM*. Richard D. Irwin, Inc., 1964.

19. Fox, Geoffrey C. and others. *Solving Problems on Concurrent Processors: General Techniques and Regular Problems, 1*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc, 1988.

20. Fujimoto, Richard M. "Performance Measurements of Distributed Simulation Strategies." *Proceedings of the SCS Multiconference on Distributed Simulation19*. Simulation Series. 14–20. 1988.

21. Fujimoto, Richard M. "Performance of Time Warp Under Synthetic Workloads." *Proceedings of the SCS Multiconference on Distributed Simulation22*. Simulation Series. 23–28. 1990.

22. Guasch, Antonio, editor. *Proceedings of the SCS Multiconference on Object Oriented Simulation*, San Diego, California: Simulation Councils, Inc.

23. Hartrum, Thomas C. and Brian J. Donlan. "Distributed Battle-Management Simulation on a Hypercube." *Proceedings of the SCS Multiconference on Distributed Simulation19*. Simulation Series. 3–7. 1988.

24. Herring, Charles. "ModSim: A New Object-Oriented Simulation Language." *Proceedings of the SCS Multiconference on Object Oriented Simulation*, edited by Antonio Guasch. 55–60. San Diego, California: Simulation Councils, Inc., 1990.

25. Horowitz, Ellis and Sartaj Sahni. *Fundamentals of Data Structures in Pascal*. Rockville, Maryland: Computer Science Press, Inc., 1982.

26. Intel Corporation. *iPSC/2 ADA Programmer's Reference Manual*, 1990.

27. Kushner, Edward J. "Parallel Simulation Using the iPSC/2." *Proceedings of the SCS Multiconference on Distributed Simulation22*. Simulation Series. 91–94. 1990.

28. Lamont, Gary B. and others. "Compendium of Parallel Programs for the iPSC Computers." Vol. 2, Ver. 1.5, Research, December 1990.

29. Lee, Kenneth J. and Michael S. Rissman. *An Object-Oriented Solution Example: A Flight Simulator Electrical System*. Technical Report CMU/SEI-89-TR-5, Software Engineering Institute, 1989 (AD-A219 190).

30. Lee, Kenneth J., et al. *An OOD Paradigm for Flight Simulators, 2nd Edition*. Technical Report CMU/SEI-88-TR-30, Software Engineering Institute, 1988 (AD-A204 849).

31. Lin, Yi-Bing and Edward D. Lazowska. "Optimality Considerations of 'Time Warp' Parallel Simulation." *Proceedings of the SCS Multiconference on Distributed Simulation22*. Simulation Series. 29–34. 1990.

32. Lomow, Greg and Dirk Baezner. "A Tutorial Introduction to Object-Oriented Simulation and Sim++." *1989 Winter Simulation Conference Proceedings*. 140–146. 1989.

33. McNear, Andrew. *Improved Task Scheduling for Parallel Simulations*. MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1991.

34. Misra, J. "Distributed Discrete-Event Simulation," *Computing Surveys*, *18*:39–65 (March 1986).

35. Nicol, David M. "Mapping a Battlefield Simulation onto Message-Passing Parallel Architectures." *Proceedings of the SCS Multiconference on Distributed Simulation19*. Simulation Series. 141–146. 1988.

36. Nicol, David M. "Performance Bounds on Parallel Self-Initiating Discrete-Event Simulations," *ACM Transactions on Modeling and Computer Simulations*, *1*:24–50 (January 1991).

37. O'Brien, David W. and John B. Gilmer. "Mixed Event- and Time-Stepped Parallel Simulation." *Proceedings of the SCS Multiconference on Distributed Simulation21*. Simulation Series. 197–202. 1989.

38. Pegden, C. Dennis, et al. "How Technology Limits Simulation Methodology." *1989 Winter Simulation Conference Proceedings*. 686–691. 1989.

39. Reed, Daniel A. and Allen D. Malony. "Parallel Discrete Event Simulation: The Chandy-Misra Approach." *Proceedings of the SCS Multiconference on Distributed Simulation19*. Simulation Series. 8–13. 1988.

40. Sarter, JoAnn M. *Optimal Iterative Task Scheduling for Parallel Simulations*. MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, May 1991.

41. Spicer, Kelly L. *Mapping an Object-Oriented Requirements Analysis to a Design Architecture that Supports Design and Components Reuse*. MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1990.

42. Stilian, Gabriel N. and others. *PERT: A New Management Planning and Control Technique*. New York: American Management Association, 1962.

43. Su, Wen-King and Charles L. Seitz. "Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm." *Proceedings of the SCS Multiconference on Distributed Simulation21*. Simulation Series. 38–43. 1989.

44. Wieland, Frederick, et al. "Implementing a Distributed Combat Simulation on the Time Warp Operating System." *The Third Conference on Hypercube Concurrent Computers and Applications2*. 1269–1276. January 1988.

45. Yu, Qing, et al. "Time-driven Parallel Simulation of Multistage Interconnection Networks." *Proceedings of the SCS Multiconference on Distributed Simulation21*. Simulation Series. 191–196. 1989.